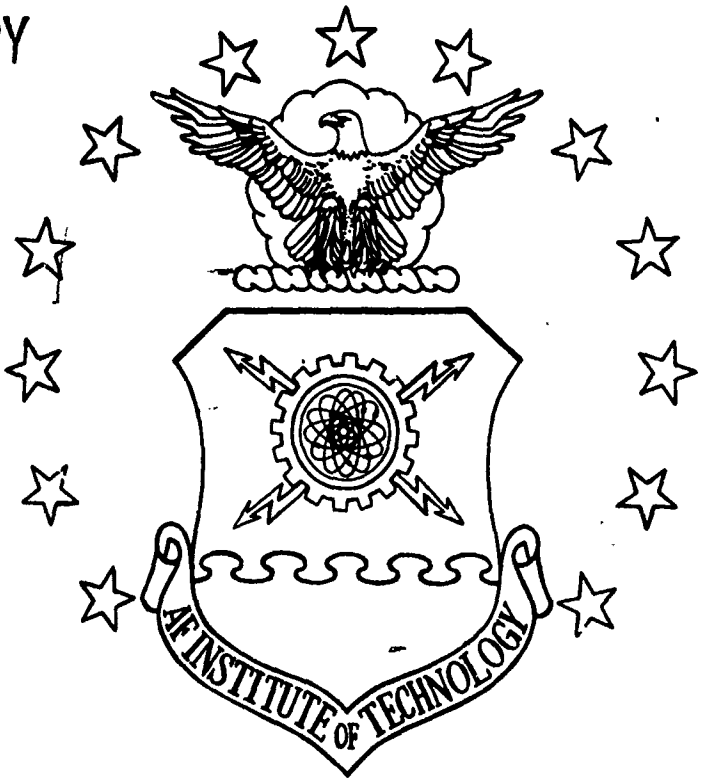


FEASIBILITY ANALYSIS OF DEVELOPING A FORMAL PERFORMANCE MODEL OF Ada TASKING

THESIS

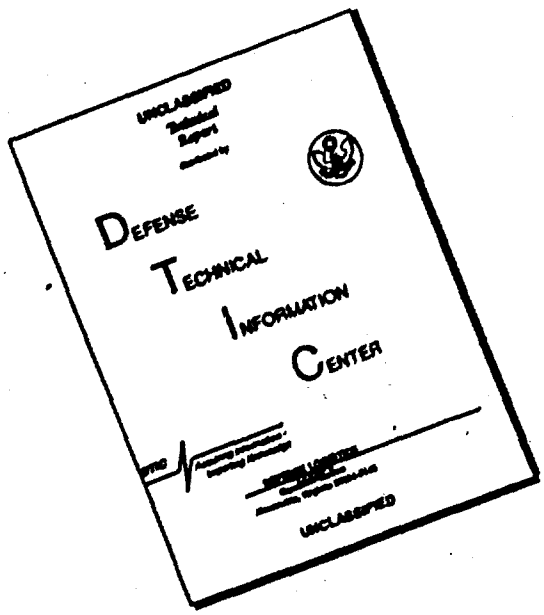
Kathryn J. Edwards
GS-12, USAF

AFIT/GE/ENG/90D-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AFIT/GE/ENG/90D-18

FEASIBILITY ANALYSIS OF DEVELOPING A FORMAL PERFORMANCE MODEL OF Ada TASKING

THESIS

Kathryn J. Edwards
GS-12, USAF

AFIT/GE/ENG/90D-18

AFIT/GE/ENG/90D-18

FEASIBILITY ANALYSIS OF DEVELOPING A FORMAL PERFORMANCE MODEL OF Ada TASKING

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Electrical Engineering)

Kathryn J. Edwards, B.S.E.E.

GS-12, USAF

December, 1990

Acknowledgments

I wish to thank my thesis advisor, Maj Paul Bailor, for helping me complete my research and thesis. I would also like to thank my committee members, Dr Thomas Hartrum and Maj Patricia Lawlis, for their time.

Additionally, I wish to thank Ken and Megan Gillam for their friendship and support during my time at AFIT. And I would like to extend a special thank you to Megan for being a "gentle" reader when reviewing my thesis document.

Finally, I would like to praise God for giving me the strength to complete this research effort.

Shout for joy to the Lord, all the earth.
Worship the Lord with gladness; come before Him with joyful songs.
Know that the Lord is God.
It is He who made us, and we are His;
we are His people, the sheep of His pasture.
Enter His gates with thanksgiving and His courts with praise;
give thanks to Him and praise His name.
For the Lord is good and His love endures forever;
His faithfulness continues through all generations.

– Psalm 100

Kathryn J. Edwards

Table of Contents

List of Figures

AFIT/GE/ENG/90D-18

Abstract

As software system requirements become more complex, software engineers must carefully design the systems to ensure the systems adequately meet all the requirements, both functional and non-functional. Because real-time systems have timing constraints, in addition to the more traditional behavioral constraints, a comprehensive software design analysis model is required which incorporates performance, timing, and behavioral constraints. Although the Ada language tasking constructs are compiler independent, Ada tasking is dependent on its runtime environment; therefore, a formal model of Ada tasking and its associated runtime environment is important in order for system designers to make realistic decisions when modeling Mission Critical Computer Resources (MCCR) systems. The main focus of this ~~research effort~~ THESIS is to determine the feasibility of developing a parameterized, formal model of Ada tasking and the associated runtime environment. This research shows that such a parameterized model can be developed using a mathematical model which incorporates real-time scheduling and queueing theory. This model can be used in the future to develop a design analysis environment for real-time embedded software systems that require Ada as the target language. Thus, given a specification for such a system, the design analysis environment can be used to obtain the information needed to support Ada software design decisions.

List of Tables

FEASIBILITY ANALYSIS OF DEVELOPING A FORMAL PERFORMANCE MODEL OF Ada TASKING

I. Introduction

1.1 Background

The Department of Defense (DoD) sponsored the development of Ada in order to combat increasing software complexity, especially in embedded, real-time computer systems. Embedded applications tend to be large, long-lived, subject to continuous change, subject to hardware constraints, and are required to be highly reliable and fault tolerant (3:15). The DoD has recently mandated Ada as the "single, common, computer programming language for Defense computer resources used in intelligence systems, for the command and control of military forces, or as an integral part of a weapon system" (7:2). These systems are typically large, embedded computer systems which have real-time processing constraints.

Real-time systems are divided into two groups: hard real-time systems and soft real-time systems. "In soft real-time systems, tasks are performed by the system as fast as possible, but they are not constrained to finish by specific times. On the other hand, in hard real-time systems, tasks have to be performed not only correctly, but also in a timely fashion. Otherwise, there might be severe consequences" (24:151).

"Hard real-time systems are defined as those systems in which correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced" (24:1). Because real-time systems have timing constraints, in addition to the more traditional behavioral constraints, a comprehensive software design analysis model is required which incorporates performance, timing, and behavioral constraints.

"Embedded computer systems are usually defined to be those computer systems that constitute a part of a larger system whose primary function is other than computational.

The primary function of the computers are to monitor and control devices" (2:3). In addition, Embedded Computer Systems (ECS) generally have real-time processing constraints which require concurrent computation. Examples of ECS in the DoD are large systems, such as, the flight control computers for the F-15, F-16, and F-111. These systems must be reliable, fault tolerant, and easy to modify over their long life span (3:15).

1.2 Statement of the Problem

The biggest problem with existing models of concurrent/parallel computation, such as Communicating Sequential Processes (CSP) and Petri Nets, is that they concentrate on modeling a system's behavior and tend to ignore, or abstract away, performance and timing issues.

It is important to specify timing requirements in the system specification for both the software and hardware. In the past, the system specification was mainly concerned with describing aspects of hardware architecture (e.g., speed and memory capacity); the software timing and speed were simply a function of the hardware and the programmer's creativity. Due to the stringent timing requirements of real-time systems, it is imperative that the system specification include both hardware and software constraints. Performance and timing requirements are critical issues for real-time systems and must be included in the system specification to determine how the specified Ada runtime environment (RTE) impacts a given design and implementation.

In order to adequately model a Mission Critical Computer Resources (MCCR) system, both performance and behavior must be described by a comprehensive formal model. Such a model of Ada tasking, and its associated RTE, is important in order for system designers to make realistic decisions when modeling MCCR systems. Although the Ada language tasking constructs are compiler independent, actual Ada tasking behavior is dependent on its RTE. It is, therefore, important for the systems and software engineers to be aware of how the RTE behaves in order to properly design these complex MCCR systems. The need for a formal model of Ada tasking and its associated RTE is increasing as Ada's usage increases in concurrent/parallel computing systems.

Because each RTE is different, an Ada tasking model needs to be parameterized to reflect the underlying RTE. Thus, the model will also be useful in determining which RTE best suits the needs of a particular embedded computer system. Alternately, since the selection of the RTE is often determined by computer system engineers, the model may be used by software engineers to point out potential problems with the existing RTE and the proposed software design.

The primary goal of this research effort is to analyze and develop a parameterized, formal model of Ada tasking and the associated RTE that incorporates the performance aspect. This goal is based upon the hypothesis that such a formal model can be developed which combines graphical and mathematical notations.

1.3 Summary of Current Knowledge

Although Ada has been mandated for embedded systems (7:2), there is doubt among members of the software engineering community as to whether Ada is capable of providing adequate support for real-time embedded computer systems (14:494–495). Nevertheless, Ada contains the tasking facilities and low-level I/O necessary for implementing real-time embedded computer systems. However, the diversity of the scheduling algorithms in each RTE causes each environment to be different and, until a standard RTE exists, software engineers must search for the environment that is appropriate for their application or create a design that is appropriate for the environment.

Concurrent programming is important for real-time systems because it is possible for events to arrive that must be handled simultaneously. Although most current programming languages only allow sequential execution, the Ada tasking facility allows programs to execute concurrently. "Tasking is an important aspect of many embedded systems ... However, tasking seems to have been neglected in most languages in production use for such systems" (15:269). The concurrent execution of tasks also makes programs more difficult to write and causes the RTE to be more difficult to implement.

1.3.1 **Ada Tasking.** "A task is the scheduling entity in a system" (24:153). The Ada Language Reference Manual (LRM) defines Ada tasks as "entities whose executions

proceed in parallel" (8:9–1). Although tasks are able to operate concurrently, there is no requirement that they must execute at the same time. A uniprocessor system may only have one process (or task) executing at any given time; the processes take turns executing and, although only one task is actually executing at a time, they are all said to be "logically" executing.

Ada tasks operate independently except when they need to synchronize with another task at which time they are said to "rendezvous" (8:9–1) (15:306). One task calls another task by issuing an entry call and when the called task accepts this call, the two tasks are in a rendezvous and may then exchange data. After completing the rendezvous, the tasks again execute independently and asynchronously.

It is possible for several tasks to call another task at the same time. When this occurs, the calling tasks are placed into an entry queue and the called task will rendezvous with the calling tasks in the queue according to a First–Come–First–Served scheduling algorithm (8:9–9) (15:276).

1.3.2 **Ada Runtime Environments.** When the first programs were written for mainframe computers, software developers created code segments for the bare computer hardware. As time went on, the software engineers agreed on basic conventions in order that their code might work together. They also built subroutines which could be reused from application to application, thus, greatly simplifying programming. These conventions and subroutines allowed the software engineer to abstract one level away from the bare machine and became a basic RTE.

At this point, however, the bare machine was still accessible to the programmer whose code would interface with both the RTE and the bare machine. This allowed each programmer to create his own abstraction of the computer. As time went on, the basic subroutines of the RTE were refined and improved with the machine–dependent features becoming the operating system and the language–dependent features being handled by the compiler.

The RTE allows the programmer to abstract away low–level implementation details which are unique to each machine. The convenience of using the RTE offsets the slight

decrease in performance due to the overhead of the RTE. Each RTE is developed for a specific machine; therefore, an application tailored for one machine may perform much differently on another machine; this is due entirely to its RTE (1:11).

Ada is defined in the LRM and any Ada RTE must comply with the requirements therein. However, the LRM allows great flexibility in how the RTE will support Ada and since there is currently no standard Ada RTE, there can be many interpretations and differing Ada RTEs (1:14).

1.3.3 **Scheduling Algorithms.** A scheduler decides the order of execution for tasks on a central processing unit (CPU), entry queue, or input/output processor (IOP). The CPU and entry schedulers are important within Ada task scheduling. Tasks may be periodic or aperiodic, independent or synchronous. Periodic tasks repeat after a fixed interval of time, whereas, aperiodic tasks occur only once, or at random intervals. Periodic tasks have a specified repetition rate called the frequency or request rate. There are varying degrees of synchronous tasks; synchronous tasks may be totally dependent on other tasks or they may only need to exchange data occasionally. As mentioned previously, Ada tasks exchange data by synchronizing in a rendezvous. Independent tasks do not need to exchange information with other tasks and, therefore, do not rendezvous with other tasks.

Deadlock, starvation, and task set performance are important issues for scheduling algorithms. Starvation occurs when a task or group of tasks is not allowed to execute. Performance is a measurement of throughput and turnaround time. Throughput measures how many tasks complete in a given time period. Turnaround time measures how long a particular task takes to complete. Each scheduling algorithm has different performance; some algorithms that allow starvation may actually have better average performance than algorithms that do not allow starvation (23:Ch 4).

Some examples of possible scheduling algorithms are: First-Come-First-Serve (FCFS), Shortest-Job-First (SJF), Round Robin (RR), and Rate Monotonic (RM), and each of these are summarized below.

1.3.3.1 **First-Come-First-Serve (FCFS).** The FCFS scheduling algorithm is the easiest to implement. New tasks are placed at the tail of the ready queue and are allocated from the head of the queue. In FCFS, all tasks are of equal priority which causes performance to be poor because the average waiting time is not minimized (18:106). The benefit of FCFS is that it will not allow starvation.

1.3.3.2 **Shortest Job First (SJF).** The SJF scheduling algorithm allocates the task with the smallest estimated execution time or "burst." If two tasks have the same burst time, then FCFS scheduling is used. There is no way of knowing what the actual length of the next burst will be without future knowledge; therefore, the next burst will be estimated upon its past performance. SJF gives the minimum average waiting time for a set of tasks since it chooses a short task before a long task. This will decrease the short task's wait time more than it will increase the long task's wait time. Thus, SJF gives a lower average waiting time. A problem with this method is that tasks with large bursts may starve since the algorithm constantly chooses the task with the shorter burst times (23:180).

Additionally, the SJF algorithm can be preemptive or non-preemptive. A preemptive SJF allows the task currently running to be interrupted when a new task arrives in the queue which has a smaller burst time. A non-preemptive SJF allows the task running on the processor to execute until it completes.

1.3.3.3 **Round Robin (RR).** The RR scheduling algorithm is similar to FCFS except that it preempts an executing task after allowing it to run for a specified time or quantum. It then places the task at the end of the ready queue. This algorithm is often used on uniprocessor systems in order to give the illusion that all the tasks are operating concurrently. The RR algorithm does not allow starvation. If the time quantum is too large, the RR algorithm behaves like the FCFS algorithm mentioned above.

1.3.3.4 **Rate Monotonic (RM).** The RM scheduling algorithm, used for a set of independent periodic tasks, selects tasks based upon their period. Tasks with shorter periods are scheduled before tasks with longer periods. "A major advantage of using the

rate monotonic algorithm is that it allows us to separate logical correctness from timing correctness concerns" (22:7).

1.3.4 **Ada RTE Schedulers.** A real-time scheduler assigns an ordering (schedule) to a set of tasks in order to meet timing, precedence, or resource requirements of real-time systems. An Ada RTE requires two schedulers: one to schedule the tasks to run on the processors; and one to schedule the synchronization points for each entry queue. These algorithms need not be the same.

Processor scheduling can use any scheduling discipline; however, certain algorithms will be more efficient for specific applications. Most Ada RTEs use FCFS or RR (26:5–15).

Each entry point has its own queue and the entry scheduler must use the FCFS algorithm, as illustrated by the following quote from the LRM.

> If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued; there is one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in the order of arrival (8:9–9).

Although the FCFS schedule does not produce the shortest schedules (i.e., best performance), it is non-preemptive. This allows the code within the accept statement to be treated as a critical section. A critical section protects a shared data area and only one task should have access to this section at a time or data may be lost or overwritten (23:83). Once an entry is selected, it must be allowed to continue executing until it is completed. No other accept statements for the tasks in the rendezvous will be allowed to execute until the current rendezvous is complete. Preemptive schedulers, such as, the RR and preemptive SJF, are not allowed because they may interrupt a task while in a critical section.

1.4 Scope

A formal model of Ada tasking is expected to generate performance statistics for a set of Ada tasks based upon specific runtime parameters, such as, the scheduling algorithm

and task execution times. The benefit of modeling the performance is that software engineers will know whether or not their design meets the required performance criteria before implementing the code. If the criteria are not met, they can modify their design until it does meet the requirements or find a RTE that meets the requirements.

The model incorporates the Ada language constructs detailed in the Ada LRM (e.g., entries, accepts, delays, priorities) (8:Ch 9) and includes the sequence of events and timing information. The actual model was developed mathematically and is based on constructs in queueing theory, set theory, and real-time scheduling. Because of its mathematical nature, this model may be difficult for non-technical people to understand; therefore, a future research effort will produce a rapid prototyping environment that will perform the mathematics and provide a simplified user interface. The ultimate goal of future research will be to create an automated tool that will return a trace of events and performance statistics.

The basic hypothesis of this research was that a formal model of Ada tasking could be developed, and that the model could be used to help develop design analysis environments for distributed real-time software systems requiring Ada as their target language. Thus, given a specification for such a system, the design analysis environment can be used to obtain the information needed to support Ada software design decisions. No effort was made to design and implement a design analysis environment in this research effort.

1.5 Approach and Methodology

In order to develop the parameterized model, a literature search was conducted to determine how and why the Ada tasking constructs were defined (8:Ch 9) (15:Ch 13) and then a survey was conducted of existing types of formal-based models of parallel/distributed computation, e.g., Petri Nets and CSP. Additionally, queueing theory and real-time scheduling were investigated. Finally, current behavioral models of real-time software analysis and design methodologies such as Real-Time Structured Analysis (RTSA) and the Design Approach for Real-Time Systems (DARTS) were explored. The goal was to ultimately incorporate the formal tasking model into one of the existing methodologies and, in fact, the DARTS methodology was selected.

After the initial research, the results were generalized into an Ada Tasking Model which synthesized the applicable concepts gleaned from the current models and methods mentioned above. The initial model provided a broad base for the final parameterized model which was developed after several iterations.

Typically, a software development life cycle contains three basic phases: a software requirements analysis phase, a software design phase, and a low–level software design and implementation phase. The model developed for this research is part of the software design phase and is concerned solely with the *Model Design Performance* block in Figure 1.1.

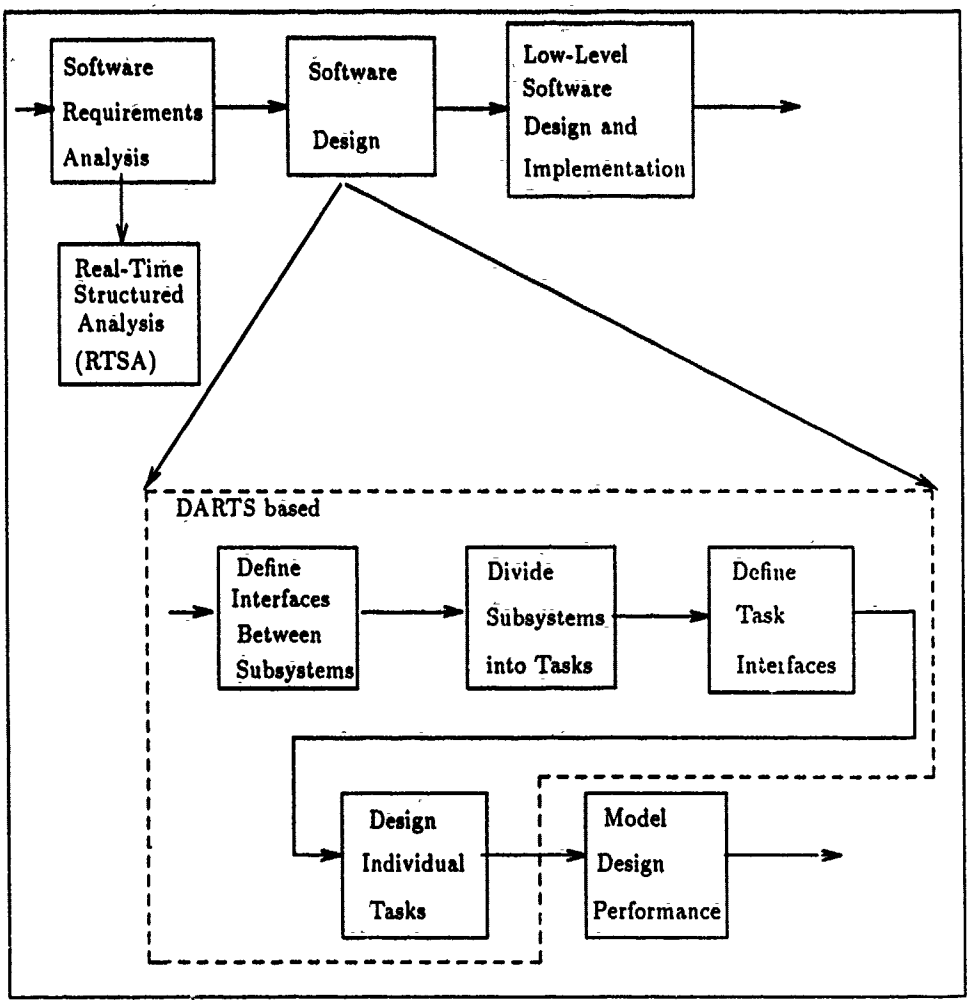


Figure 1.1. Software Development Life Cycle

This research document is organized as follows: Chapter II describes formal modelling; Chapter III defines the top-level design for a proposed performance model; Chapter IV contains a description of the detailed design of the proposed model; Chapter V discusses validating the model; Chapter VI gives recommendations for further research. There are three appendices attached: Appendix A contains a list of the acronyms used in this document; Appendix B contains a detailed Structured Analysis and Design Technique (SADT) description of the Ada tasking model; and Appendix C contains the programs used to validate the model.

II. Modeling

As software system requirements become more complex, software engineers need to carefully design their systems to ensure that the system adequately meets all its timing and behavior requirements. Modeling allows a designer to create an abstraction of the system and add detail in an iterative fashion as the requirements become more clearly understood. Due to the complexity of real-time system requirements, it is extremely difficult for a designer to initially understand the entire system and how its components interact. According to Pritsker, "[t]he entire model building approach is performed iteratively" (20:5). By modeling the system with increasing levels of complexity, a designer will gain a better understanding of the requirements and have more confidence that the design meets those requirements.

Coding is an expensive process and, once code has been written, many managers are not willing to throw it away and start over if problems are found with the design. Instead, they will encourage the programmers to manipulate the existing code to make it fit the new scenario. One way to save time, money, and wasted effort, when designing software, is to first develop the top-level design and then create a model of that design. The model will allow a designer to abstract away the low-level details until the system requirements are better understood. However, it must be remembered that an inherent problem with models is that simplifying assumptions must be made in order to abstract away unwanted or unnecessary detail. These assumptions must be valid or they invalidate the model since the model no longer accurately reflects the intended system.

2.1 Definition of Models

Models for a software design are much more flexible than the code for that design; therefore, changes can more easily be made to the model. This flexibility encourages the designer to create the model in stages; ultimately creating a software design model which accurately portrays the real system and meets the stated requirements. Modeling also allows a designer to compare multiple approaches to solving a problem. Therefore, a designer can confidently choose to implement the design which represents the optimum

solution. Once the model has been used to analyze a proposed software design, the code can be written and the designer will have confidence that the code will accurately reflect the requirements of the system.

There are several types of models: descriptive (natural language); physical (actual representation); symbolic language (mathematical); graphical; and procedural (simulation) (9:6). The models of interest to this research are the symbolic and graphical models. The symbolic models are concise, formal models which mathematically describe a system's behavior. Graphical models describe the behavior of a system pictorially.

This chapter presents an overview of the following formal models: Communicating Sequential Processes (CSP) (13), Petri nets (19), and Unbounded Nondeterministic Iterative Transformations (UNITY) (4). Additionally, three graphical models are described: Real-Time Structured Analysis (RTSA) (2), Design Approach for Real-Time Systems (DARTS) (11), and Structured Analysis and Design Technique (SADT) (12); and the top-level design of the Ada tasking model is introduced.

2.2 Formal Models of Parallel/Distributed Computation

2.2.1 **Communicating Sequential Processes (CSP).** CSP, a formal model developed by C.A.R. Hoare (13), can be used to model event driven systems. CSP had a strong influence upon the design of the Ada rendezvous; however, while the rendezvous in Ada is one-sided, or asymmetric, the communication between tasks in CSP is symmetric (15:306-308). The asymmetry of Ada task communication allows one task to call another task, such that, the called task does not know the name of the task which is calling it. The result of the asymmetry is that entry queues may be formed.

Processes in CSP can execute concurrently by communicating via message passing; although processes in CSP can execute concurrently, only one event is allowed to occur at a given time. Hence, it is not possible to determine if two events happened simultaneously. If strict concurrency is necessary, it must be modeled as a single-event occurrence.

2.2.2 **Petri Nets.** Petri nets combine graphical and mathematical notations. As with CSP, it is not possible to model simultaneous events (19:37). The execution of a Petri

net is nondeterministic (19:36) and general Petri nets abstract away timing issues: "There is no inherent measure of time or the flow of time in a Petri net" (19:35). In addition, a major disadvantage of Petri nets is that the complexity of the model increases with the size of the system. This increased complexity means that they tend to be useful only for manually modeling small systems.

2.2.3 **UNITY.** UNITY, developed by Chandy and Misra, is a "computational model and a proof system" (4:8). The goal of UNITY is to mathematically design programs, at a high-level, which are free from implementation issues, such as computer architecture and language, and whose correctness can be proven. The disadvantages of UNITY are that it is hard to understand and proving that the high-level UNITY program meets the requirements does not mean that the program implementation meets the requirements. Another disadvantage of UNITY is that it abstracts away timing issues and does not allow a designer to specify a control sequence. Thus, timing issues cannot be modeled in UNITY.

The formal models mentioned in this section, i.e., CSP, Petri nets, and UNITY, are adequate for modeling the behavior of the system; however, they ignore timing requirements and are difficult to apply. The next section discusses three graphical models.

2.3 Graphical Models

2.3.1 **Real-Time Structured Analysis.** RTSA, a variation of structured analysis and design developed by Yourdan and DeMarco, is used during the software requirements analysis phase. (See Figure 1.1 for the Software Life Cycle.) RTSA extends the traditional data flow diagrams to include timing information through the use of control flows and transforms. The designer creates the data flow/control flow diagrams during RTSA and supplements the diagrams by natural language or state transition diagrams. The disadvantage of RTSA is that it has no formal mathematical basis that can be used to analyze the resulting RTSA design.

2.3.2 **Design Approach for Real-Time Systems.** DARTS is also an extension of structured analysis and design. Using a RTSA input, DARTS focuses on decomposing

a system into a set of concurrent tasks and models the inter-task communication. After DARTS is completed, each subsystem has been divided into sets of tasks which operate concurrently and each task has a single thread of control. However, DARTS does not specify the timing, hardware, or RTE requirements, and as with RTSA, the DARTS design has no mathematical basis for evaluating the quality of the design.

DARTS is performed after the software requirements analysis has been completed and it has four steps (represented within the dashed lines in Figure 2.1):

- define the interfaces between the subsystems;
- structure the subsystems into parallel tasks;
- define the interfaces between the tasks; and
- design individual tasks using structured design.

2.3.3 **Structured Analysis and Design Technique[1].** SADT (12), as the name suggests, is also a variation of structured analysis and is used during the software requirements phase. (See Figure 1.1 for the Software Life Cycle.) As with RTSA and DARTS, an SADT design has no formal mathematical basis and cannot be proven mathematically. SADT is described here in detail because SADT was used in the requirements analysis, specification, and design of the developed Ada tasking model.

A generic SADT diagram is shown in Figure 2.2.

2.3.3.1 **Interfaces.** The basic element in SADT is the function which describes a process or action and is represented by a box. The arrows define interfaces and are described in the following quote.

> Interfaces are represented by arrows entering or leaving the box. The type of interface is indicated by the side of the rectangle to which it is connected ... *input* arrows enter the left side of the function box, *output* arrows leave the right side of the box, *control* arrows enter the top of the box, and *mechanism*

[1]SADT is a trademark of SofTech.

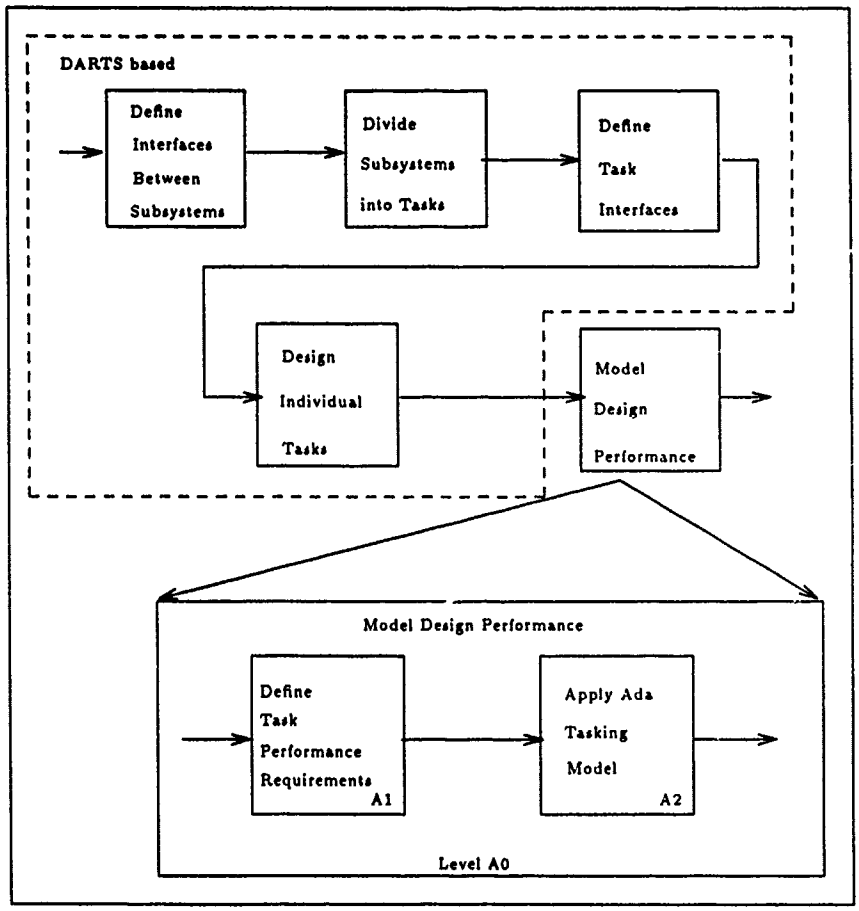


Figure 2.1. Software Design

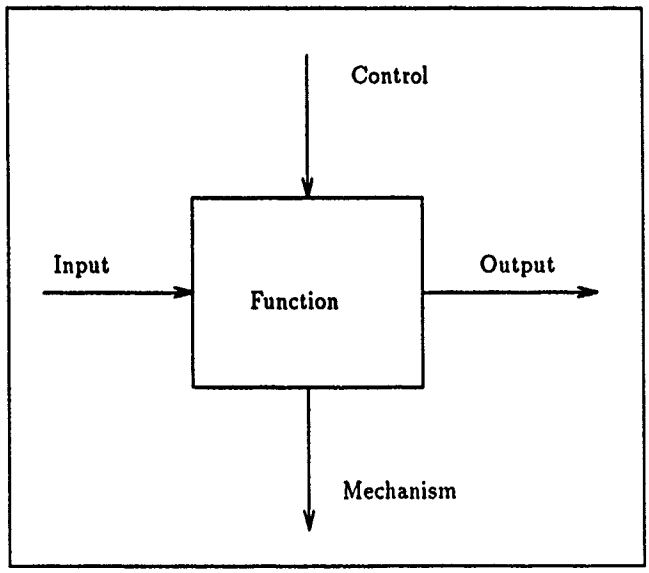


Figure 2.2. Generic SADT Diagram

> arrows either enter or leave the bottom of the box. The function is viewed as transforming its inputs into outputs under the guidance of its controls. (12:7)

SADT descriptions do not impose timing requirements so the arrows merely represent constraints; a function cannot commence unless its control and inputs are available. "The functions represent processes that must occur, but may in fact occur simultaneously. The arrows represent data or information produced by or needed by a function. They should not be viewed as flows or sequences of operations" (12:7).

Additionally, every function requires at least one control arrow and one output arrow, regardless of whether or not there are any input or mechanism arrows. The mechanism arrows "indicate a means of performing the function" (12:7).

2.3.3.2 **Hierarchy of Numbering.** SADT has a special numbering system which denotes the hierarchical level of decomposition. Each function box label begins with an "A" which stands for "Activity." The top-level, or environment model, is labeled as level A-0 and contains a single box. This box shows the interconnection of the system to be modeled with its environment. The first level of decomposition is labeled A0 and represents the major subfunctions of the system. Each of the boxes, or functions, at level

A0 are sequentially labeled A1, A2, etc. In turn, the functions on the level A1 diagram are labeled A11, A12, etc. This numbering system facilitates determining the level of the diagram and maintaining consistency between the levels.

2.4 Deficiencies of Real-Time Models

The formal models described in Section 2.2 model the behavior of the system but are either too complex to use, ignore timing requirements, or both. The graphical models described in Section 2.3 lack the formalism of the formal models which is important when analyzing the design.

Because real-time systems have these timing constraints, in addition to behavioral constraints, a formal software design analysis model is required which incorporates performance, timing, and behavioral constraints. Consider, for example the DARTS design methodology, the box labeled *Model Design Performance* in Figure 2.1 is an additional step added to DARTS which deals with solving these deficiencies, and the object of this research is to analyze the feasibility of constructing such a model. Specifically, the next chapter introduces the top-level design for the design performance model.

III. Design of the Ada Tasking Model

This chapter describes the top-level design of the performance model of Ada tasking. The Model Design Performance activity in Figure 3.1 is performed after the initial DARTS design has been accomplished. If application of the performance model shows that the software design fails to perform as required, then the software design must be modified or reaccomplished and the performance model reapplied. This feedback loop continues until such time as it is determined the software design satisfactorily meets its performance requirements.

3.1 Model Design Performance

Figure 3.1 is an SADT diagram representing the environment, or A-0 level diagram. The function, *Model Design Performance*, is an extension to DARTS and is applied after the initial DARTS design has been completed. *Model Design Performance* requires that the software design first be produced using DARTS. In addition, *Model Design Performance* requires the non-functional requirements and scheduling information about the RTE.

The *Model Design Performance* activity is decomposed into two functions: *Define Task Performance Requirements* (A1) and *Model Ada Tasking* (A2). Note that the output arrow, *Performance Data*, has been decomposed into *schedulable, task schedule, entry trace* and *performance stats.* The decomposition for *Model Design Performance* is shown in Figure 3.2, representing the A0 level, and the activities A1 and A2 are described in the Sections 3.2 and 3.3.

3.2 Define Task Performance Requirements

The function, *Define Task Performance Requirements*, has three interface arrows: the control arrow is the software design; the input arrow is the non-functional requirements; and the output arrow is the task information which includes the task names, periods, execution times, etc. (For more detail on the task information, see the data dictionary in Appendix B.) Note that there is no mechanism for this function. The remainder of this research concentrates on developing the Ada Tasking Model (level A2) with certain

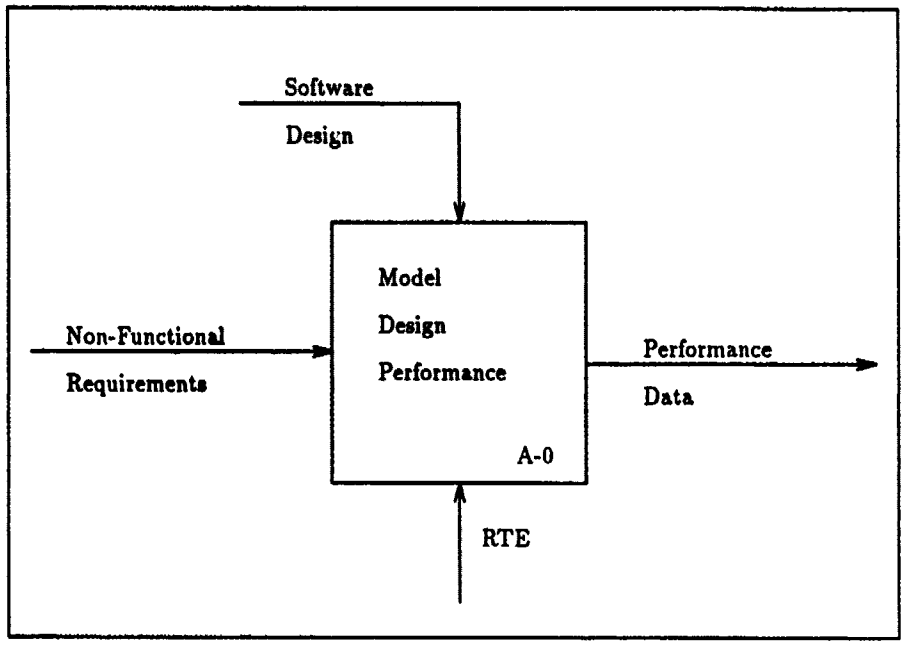


Figure 3.1. Environment Model

assumptions made about the outputs of A1. A more comprehensive examination of defining task performance requirements (A1) will be accomplished in follow-on research.

3.3 Model Ada Tasking

The function A2, *Model Ada Tasking*, has six interface arrows: one mechanism, four outputs, and one control. The mechanism arrow is the RTE. The four output arrows are: *schedulable*, which is a Boolean flag that tells the designer if the tasks can be scheduled based upon the given RTE and task information; *task schedule*, which is a possible schedule based upon the scheduling information from the RTE mechanism and the task information; *entry trace*, which is a possible sequence of entry points based upon the task schedule; and *performance statistics*, which are statistics describing the design performance. The control arrow is the task information which is the same as the output arrow from function A1.

The task information includes the task name, execution time, period, etc. In order to schedule a group of tasks, the task execution time (E_i) and period (T_i) must be known. These values are used to determine each load factor which is the execution time of a task

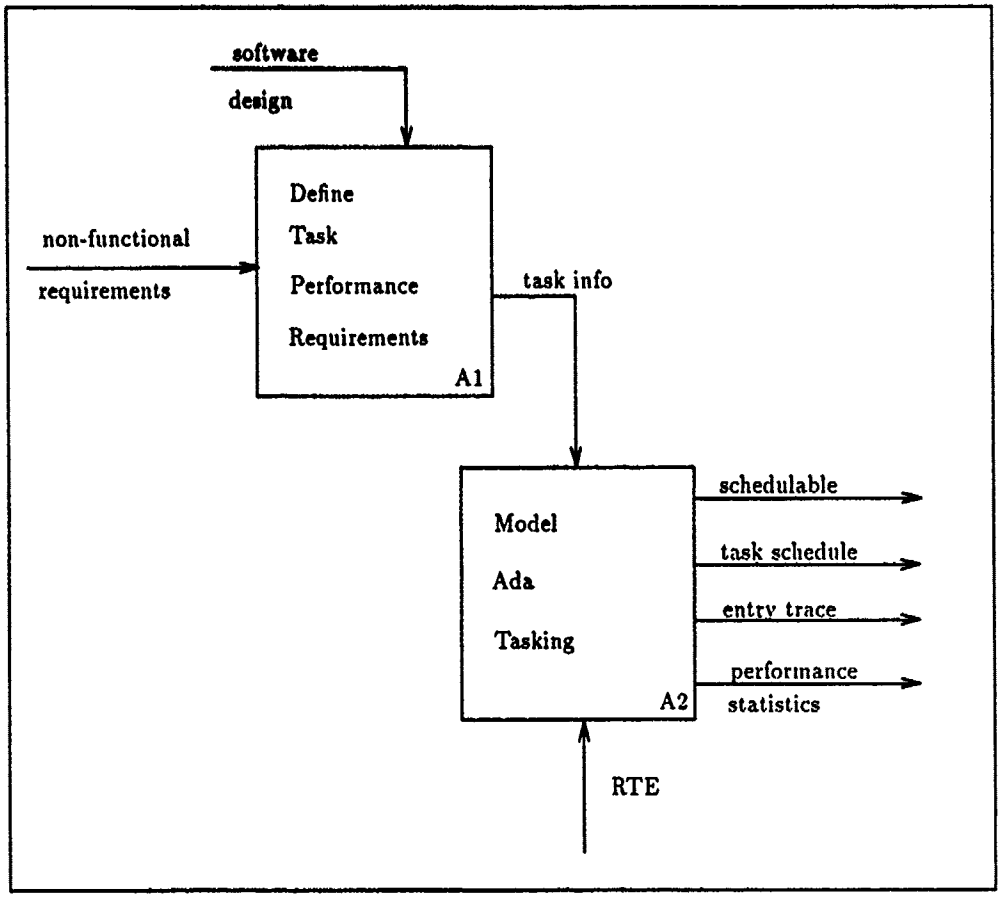


Figure 3.2. Level A0

divided by its period ($\frac{E_i}{T_i}$). Each processor may only execute a set of tasks if the sum of their load factors is less than 1, where 1 represents 100% utilization of the processor.

The basic components of the Ada tasking model (shown in Figure 3.3) are *Determine Unschedulability, Create Schedule*, and *Model Entry Calls.* Functions A21 and A22 determine if the given set of tasks are schedulable and, if so, a schedule is found. Functions A21–A23 are summarized below; the details of these functions have been placed in Appendix B.

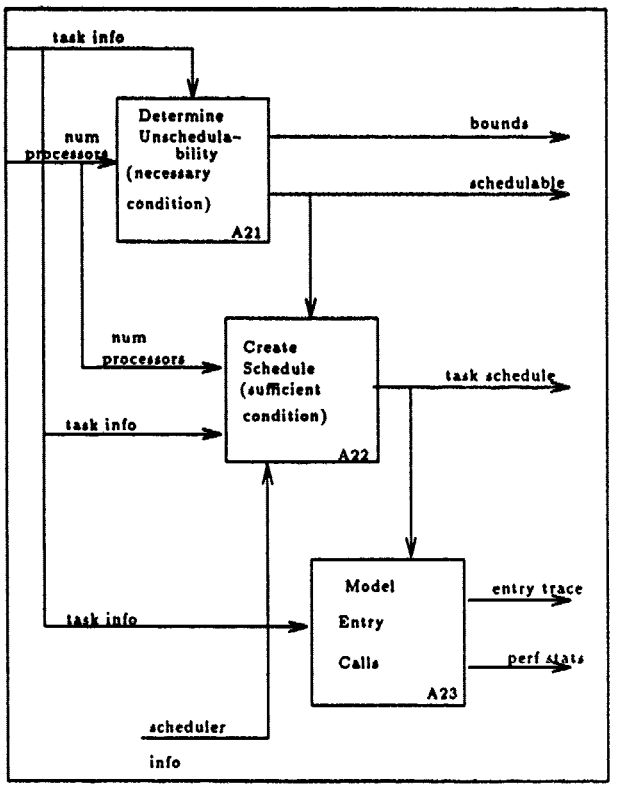


Figure 3.3. Level A2

3.3.1 **Determine Unschedulability.** Function A21, *Determine Unschedulability*, has four interface arrows: a control arrow labeled *task info*; an input arrow labeled *num processors*; and two output arrows labeled *bounds* and *schedulable*. The interface arrows, *task info* and *schedulable*, were defined above. The interface arrow, *num processors*, refers to the number of processors which will be used in the target machine for the system

the designer is modeling. The interface arrow, *bounds*, is an integer range which defines the upper and lower bound on the number of processors required for the target machine. If *num processors* is within this range and if *schedulable* is true, then the designer can proceed to Level A22. If either of these conditions is false, then the designer must either redesign the software tasks (i.e., reapply DARTS) or change the number of processors in the system specification. Redesigning the software is the logical first step; changing the system specification should only be done as the last resort.

Note that function A21 only determines if the given set of tasks **cannot** be scheduled and does not guarantee that the tasks are, in fact, schedulable. Hence, the algorithm used to determine unschedulability is a necessary condition, not a sufficient condition. The only way to know for certain if the tasks are schedulable is to apply function A22 and actually create a schedule of the tasks.

3.3.2 **Create Schedule.** Function A22, *Create Schedule*, is performed after the design passes Level A21. This function has five interface arrows: a control arrow labeled *schedulable*; two input arrows labeled *num processors* and *task info*; a mechanism arrow labeled *scheduler info*; and an output arrow labeled *task schedule*. The interface arrows, labeled *schedulable, num processors* and *task info* are defined above. The mechanism arrow, *scheduler info*, refers to the type of task scheduler used in the RTE. The output arrow, *task schedule*, is the schedule of tasks which were developed in the DARTS design.

3.3.3 **Model Entry Calls** Function A23, *Model Entry Calls*, has four interface arrows: a control arrow labeled *task schedule* which is a schedule of the tasks which were developed in the DARTS design; an input arrow labeled *task info* which includes the task names, periods, execution times, etc.; and two output arrows labeled *entry trace* and *performance stats*. The *entry trace* is a sequence of the entry points and the *performance stats* are statistics describing the design performance.

The next chapter describes *Model Entry Calls* in detail after providing background information on Ada entry points and arrival distributions.

IV. Model Entry Calls

4.1 Ada Entry Points

There are two main distinctions among entry points. The first distinction is between single entries and entry families. A single entry queues its calls according to the FCFS discipline. An entry family queues its calls according to the index associated with the call, where the index can denote the priority of the call. Entry families represent a hierarchy of queues; each index has its own entry queue which uses the FCFS discipline. Thus, calls can be accepted from the queues in the order of their index which allows a priority scheme to be developed.

The second distinction is among timed, conditional, and simple entries. A timed entry allows balks; this call is cancelled if the rendezvous does not begin within the specified time. A conditional entry is a special case of the timed entry with the time limit set to zero. The call is cancelled unless the rendezvous can occur immediately. The simple entry can be thought of as a timed entry with a time limit of infinity. A simple call is not revoked once it has been issued.

Combining the above categories gives six types of entry points: single timed, single conditional, single simple, family of timed, family of conditional, and family of simple entries. Table 4.1 describes the different types of entries and how they will be modeled.

4.2 Modeling Assumptions

The interarrival and service times will be modeled by the exponential distribution based on the following assumptions:

- the current arrival/service time is independent of the last arrival/service; and
- the arrival/service time is independent of the number in the entry queue.

These assumptions appear to be valid for the Ada Tasking Model because the time since the last arrival and the time in service refer to real or continuous time and not to computer processing time which may be affected if the task is swapped out of the

Table 4.1. Modeling Entry Queues

Entry Type	Queueing Model	Service Algorithm	Balks
single simple	M/M/1	FCFS	no
single timed	M/M/1	FCFS	yes
single conditional	M/M/1	FCFS	yes
family simple	M/M/m	FCFS	no
family timed	M/M/m	FCFS	yes
family conditional	M/M/m	FCFS	yes

processor. Additionally, these assumptions are frequently used, as demonstrated by the following quote from Trivedi (25:114):

> Thus the following random variables will often be modeled as exponential:
>
> 1. Time between two successive job arrivals to a computing center (often called interarrival time).
>
> 2. Service time at a server in a queuing network; the server could be a resource such as the CPU, I/O device, or a communication channel.

Therefore, the entry queues will be modeled using M/M/1 queues. The term "M/M/1" comes from Kleinrock (16) who uses the notation "A/B/m/K/M" to describe queueing systems. The "A" represents the arrival distribution and "B" represents the service distribution. The following quote from Kleinrock further explains the notation.

> *m*–Server queue with $A(t)$ and $B(x)$ identified by A and B, respectively, with storage capacity of size K, and with customer population of size M (if any of the last two descriptors are missing they are assumed to be infinite) (16:399)

In this instance, the queues are single servers with the interarrival distribution, $A(t)$, and service distribution, $B(x)$. Both of these distributions are described as M which means

that they denote the exponential distribution. The Probability Distribution Function (PDF) of the exponential distribution is (16:65):

$$X(t) = 1 - e^{-\lambda t} \text{ for } t \geq 0 \tag{4.1}$$

The Probability Density Function (pdf) is, therefore (16:65):

$$x(t) = \frac{d(X(t))}{dt} = = \lambda e^{-\lambda t} \text{ for } t \geq 0 \tag{4.2}$$

The exponential distribution is especially interesting due to its memoryless property which states that, "the past history of a random variable that is distributed exponentially plays no role in predicting its future" (16:66). This property is represented by the following equation, where,

$$P[X \leq t + s \mid X > s] = P[X \leq t] \tag{4.3}$$

Thus, the arrival time is independent of when the last arrival occurred and the service time is independent of the time already spent in service. Another assumption is that the system is in steady state.

The assumption of M/M/1 queues also keeps the solution of the queueing network tractable, as illustrated by the following quote:

> When one relaxes the Markovian assumptions on arrivals and/or service times, then extreme complexity in the interdeparture process arises not only from its marginal distribution, but also from its lack of independence on other state variables. (16:155)

4.2.1 **Arrival Distributions.** Calls arrive at the entry queues according to either a random or general distribution. (For the purposes of this research, a general distribution refers to a non-random distribution, e.g., a deterministic distribution.) In addition, calls may or may not repeat. Repeating, or cyclic, calls may occur with either a random or

general distribution, both of which may repeat at a known frequency. Non-repeating calls are also based upon random or general distributions and can be thought of as repeating calls with an infinite period.

Random calls can be made from either the hardware or the software; general calls can only be made from the hardware because Ada offers no timing guarantees for the software entry calls. Additionally, even though the hardware can make entry calls at a specified time, the Ada tasking facility accepts these calls in an arbitrary manner (8:9-13) (15:285). Therefore, arrivals to the entry queues are assumed to be random.

4.2.2 **Service Distributions.** Regardless of the arrival pattern, the service time of the entry calls is random because of the inherent randomness (nondeterministic nature) of select statements within Ada's tasking facility. "If several alternatives can be selected, one of them is selected arbitrarily" (8:9-13).

The assumption of random service is not as crucial as the assumption of random arrivals. M/G/1 queues can still be modeled using the M/M/1 queueing network equations which are used in this research (17:226).

4.2.3 **General Distributions.** If the Ada tasking facility were to be changed in the future so that it incorporated timing guarantees, then the following modifications would be necessary to the model: model general arrival times with the G/M/1 queue: model general service times with the M/G/1 queue; and model both general arrivals and service times by the G/G/1 queue.

4.3 Entry Call Model

Figure 4.1 depicts level A23 of the Ada Tasking Model design. This level has six activities: *Get Entry Precedence Requirements; Create Entry Trace; Create Network of Entry Queues; Model Arrival Patterns; Solve Network Equations*; and *Gather Performance Statistics.* The following sections provide a summary of the six functions. For more details, see Appendix B.

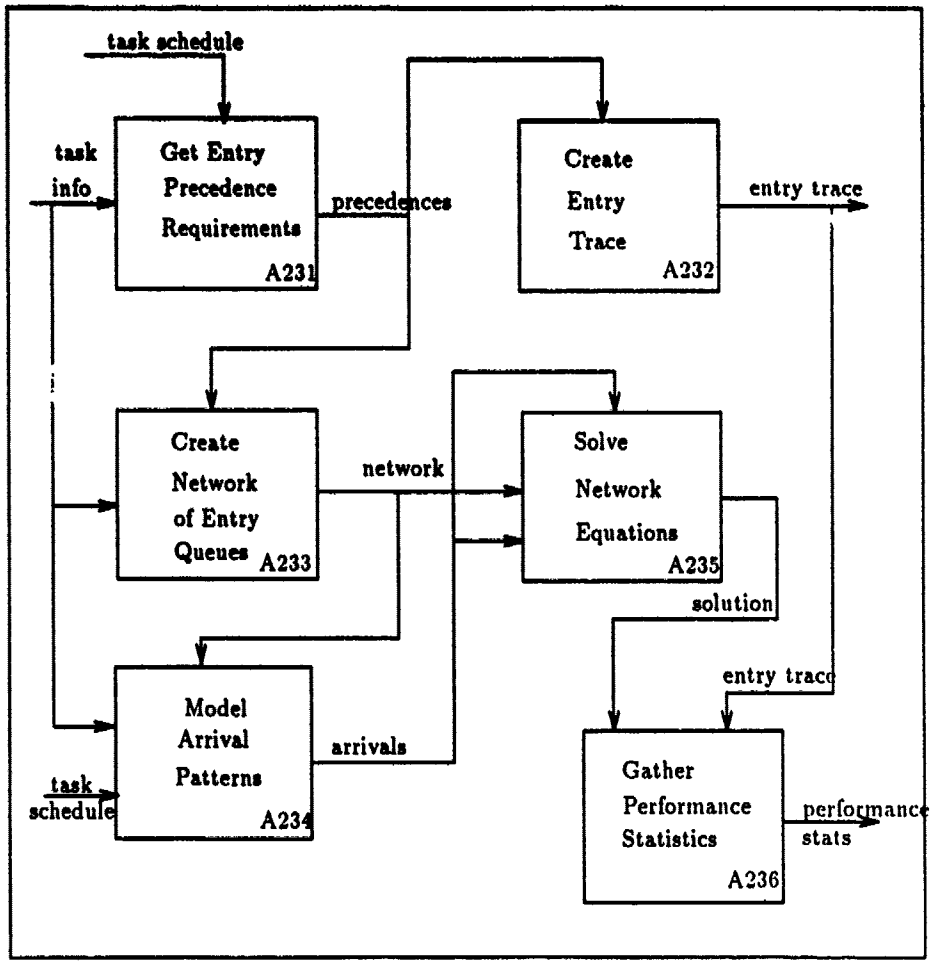


Figure 4.1. Model Entry Calls – Level A23

4.3.1 **Get Entry Precedence Requirements.** Function A231, *Get Entry Precedence Requirements*, has three interface arrows: a control arrow labeled *task schedule*; an input arrow labeled *task info*; and an output arrow labeled *precedences*. The arrow *task schedule* is a schedule of the tasks designed in DARTS. The task information includes the task names, periods, execution times, precedences, etc. The arrow *precedences* refers to the NxN matrix of dependencies which is developed from the precedence information contained in *task info*.

The precedence requirements are input by the designer in the form of *task info*. The precedences are transferred to an NxN matrix, where N is the number of entry points. The matrix contains Boolean values, with dependencies denoted by TRUE. The function, *Get Entry Precedence Requirements* reads in the precedence requirements and outputs an NxN matrix depicting those precedences.

4.3.2 **Create Entry Trace.** Function A232, *Create Entry Trace*, has two interface arrows: a control arrow labeled *precedences*, which is output from function A231 (Section 4.3.1) and an output arrow labeled *entry trace*, which is a sequence of entry points.

The function, *Create Entry Trace*, creates one possible sequence of entry calls based upon the precedence matrix. The entry trace will be created using a CSP–like language: the alphabet consists of the entry points. Another use for the entry trace is to show whether or not deadlock occurs within the system. Section 4.3.6 details the uses of the entry trace.

4.3.3 **Create Network of Entry Queues.** Function A233, *Create Network of Entry Queues*, has three interface arrows: a control arrow labeled *precedences*, which originates from Function A231; an input arrow labeled *task info*, which was described in Section 4.3.1; and an output arrow labeled *network*, which is the queueing network. The queues represent the entry points, i.e., the accept statements. The queueing network will be modeled as a network of M/M/1 queues. The connections between the separate queues are contained within an NxN matrix Q. This matrix is similar to the precedence matrix described above except that the entries are values between 0 and 1 which denote

the probability of leaving queue i (represented by row i) and entering queue j (represented by column j).

The function, *Create Network of Entry Queues*, reads in the *precedences* matrix and the *task info* and produces a queueing network matrix depicting the interconnections between the entry queues.

4.3.4 **Model Arrival Patterns.** Function A234, *Model Arrival Patterns*, has four interface arrows: a control arrow labeled *network*, which originates from function A233; an input arrow labeled *task info*, which was described in Section 4.3.1; an input arrow labeled *task schedule*, which originates from function A22; and an output arrow labeled *arrivals*, which describes the arrival distributions for each of the queues.

The function, *Model Arrival Patterns*, assigns an arrival distribution to each of the entry queues. As previously stated, this research assumes the arrival patterns can be modeled using the exponential distribution. If future changes are made to the Ada tasking facility which nullify these assumptions, then this segment of the model will need to be redefined. However, as mentioned previously, this model is valid for either M/M/1 or M/G/1 queueing networks.

4.3.5 **Solve Network Equations.** Function A235, *Solve Network Equations*, has four interface arrows: a control and input arrow labeled *arrivals*, which originates from Function A234; an input arrow labeled *network*, which originates from Function A233; and an output arrow labeled *solution*, which is a matrix of equations that solve the queueing network. The function, *Solve Network Equations*, reads in the network matrix and the arrival distributions and finds the arrival rates (λ) and the service rates (μ).

Because the queues are assumed to be M/M/1, the network can be modeled using Jackson network equations (16:149–150). Jackson's method allows open or closed networks and feedback. The individual queues are referred to as "nodes" and the arrivals as "customers." Arrivals from outside the system arrive according to the Poisson distribution (e.g., interarrivals have exponential distributions) at the rate γ_i. Customers move from

node j to node i with probability r_{ji}. (Note that the probability of a customer leaving node j and feeding back to the same node j is r_{jj}.)

The total arrival rate of customers to node i is found by summing the outside arrivals and the internal arrivals which arrive from nodes within the network. The following equation represents the total arrival rate for node i.

$$\lambda_i = \gamma_i + \sum_{j=1}^{N} \lambda_j r_{ji} \; for \; i = 1, 2, ..., N \tag{4.4}$$

A network of N nodes will have N equations of this form.

4.3.6 **Gather Performance Statistics.** Function A236, *Gather Performance Statistics*, has three interface arrows: a control arrow labeled *solution*, which is the output from function A235; a control arrow labeled *entry trace*, which is the output from function A232; and an output arrow *performance stats*, which describes the performance statistics generated by the model developed within this thesis.

The function, *Gather Performance Statistics*, gathers and calculates queueing statistics, such as:

- arrival rate (λ)
- service rate (μ)
- utilization of queues (ρ)
- time in queue (T)
- number in queue ($\bar{N}_q$)
- service time (S)
- wait time (W)

The statistics gathered from the queueing network are averages. See Appendix B for the equations.

The entry trace generates a sequence of the entry calls. The remainder of the information gained from the entry trace depends upon the implementation of the trace. This

model assumes that the trace will be implemented as print statements within the program and that the entries will be time stamped. Thus, information gathered from the entry trace, in addition to the list of entries, is the number of times each entry was called, when the calls were made, and general information for the interarrival and service distributions.

V. Validation

This chapter contains the first attempt at validating the performance model of Ada tasking. The validation compares the results of applying the Ada Tasking Model with a queueing simulation and an Ada implementation. The Dining Philosophers problem was chosen to demonstrate the validation because it is a well-defined problem requiring concurrent programming.

The life cycle for this validation, shown in Figure 5.1, parallels the life cycle defined in Figure 1.1. The numbers in Figure 5.1 correspond to the sections where each of the phases are developed. A general application of the model is contained in Section 5.3 and Section 5.6 contains the application for a specific case.

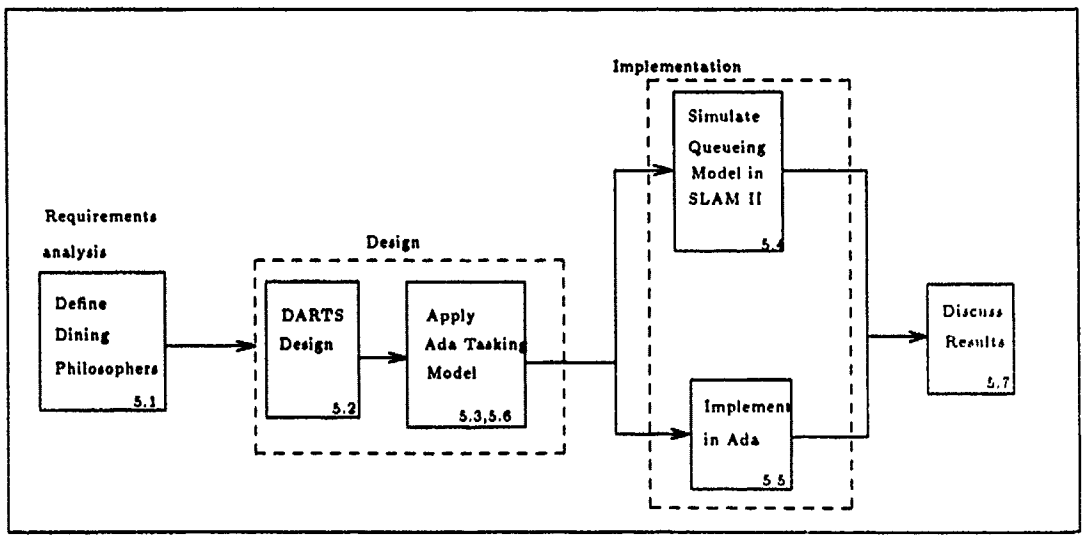


Figure 5.1. Validation Life Cycle

5.1 The Dining Philosophers

The Dining Philosophers, first presented by E.W. Dijkstra (10:83–99) (13:75–81), is a classic synchronization problem which is used to benchmark concurrent programming facilities because it illustrates deadlock and starvation problems among shared resources. Hoare describes the Dining Philosopher problem as follows:

> In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs ... To the left of each philosopher there was laid a golden fork, and in the centre stood a large bowl of spaghetti, which was constantly replenished. (13:75)

In order to eat, the philosopher picks up the forks closest to him, one at a time. Once the philosopher obtains both forks, he eats until he is no longer hungry, puts down the forks, and goes away to think until he is hungry again and then the whole process repeats. Once a philosopher has possession of a fork, he will not relinquish it until he has finished eating, causing potential problems with deadlock and starvation.

Deadlock occurs when all five philosophers decide to eat at the same time and each picks up one fork. None of the philosophers will release his fork until he has eaten, but none of the philosophers can eat until he gets another fork; the philosophers are deadlocked. Starvation is slightly different from deadlock in that one or more of the philosophers obtains both forks and eats while another is never able to pick up both forks and, therefore, starves to death.

The Dining Philosopher implementation used to validate the Ada Tasking Model employs a host to ensure that deadlock will not occur. Each philosopher must ask the host's permission to enter and leave the dining room and the host allows only four philosophers in the dining room at a time; thus, ensuring that at least one of the philosophers will be able to eat at a time. Deadlock has been avoided (10:88). Note that starvation may still occur if one of the philosophers sits at the table and never gets both forks.

5.2 Design Approach for Real-Time Systems Design

Figure 5.2 depicts the state flow for one of the philosophers in the Dining Philosophers problem. Once a philosopher enters the system he continues to eat and think until he dies. The following sections describe the DARTS design for the Dining Philosophers problem and describe the task information required to apply the tasking model.

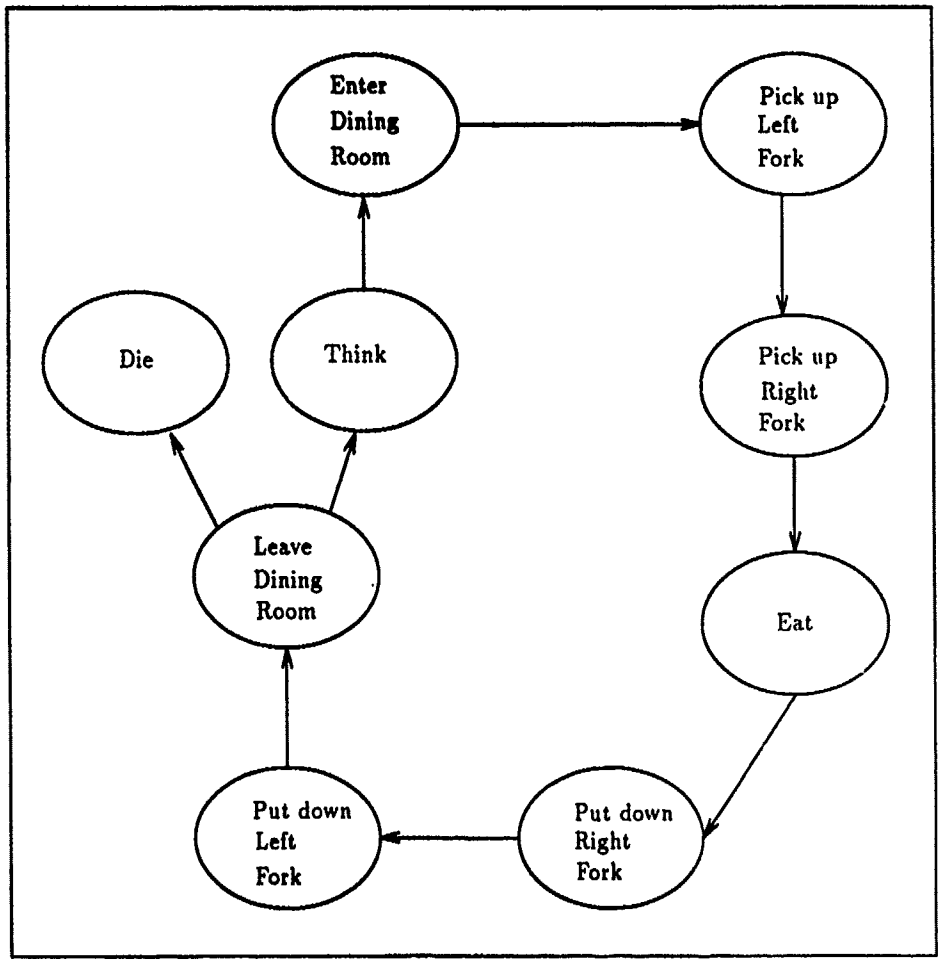


Figure 5.2. Flow Diagram for Dining Philosopher

5.2.1 **DARTS Design.** Figure 5.3 shows the DARTS diagram for one of the philosophers. ***Enter Dining Room***, and ***Leave Dining Room*** have been placed in the task ***Host***. ***Pick Up Left Fork***, ***Pick Up Right Fork***, ***Put Down Right Fork***, and ***Put Down Left Fork*** have been condensed to ***Pick Up Fork*** and ***Put Down Fork*** in task ***Fork***. The ***Philosopher*** task makes calls to ***Host*** and ***Fork***. Note that there are five fork tasks, five philosopher tasks, and one host task.

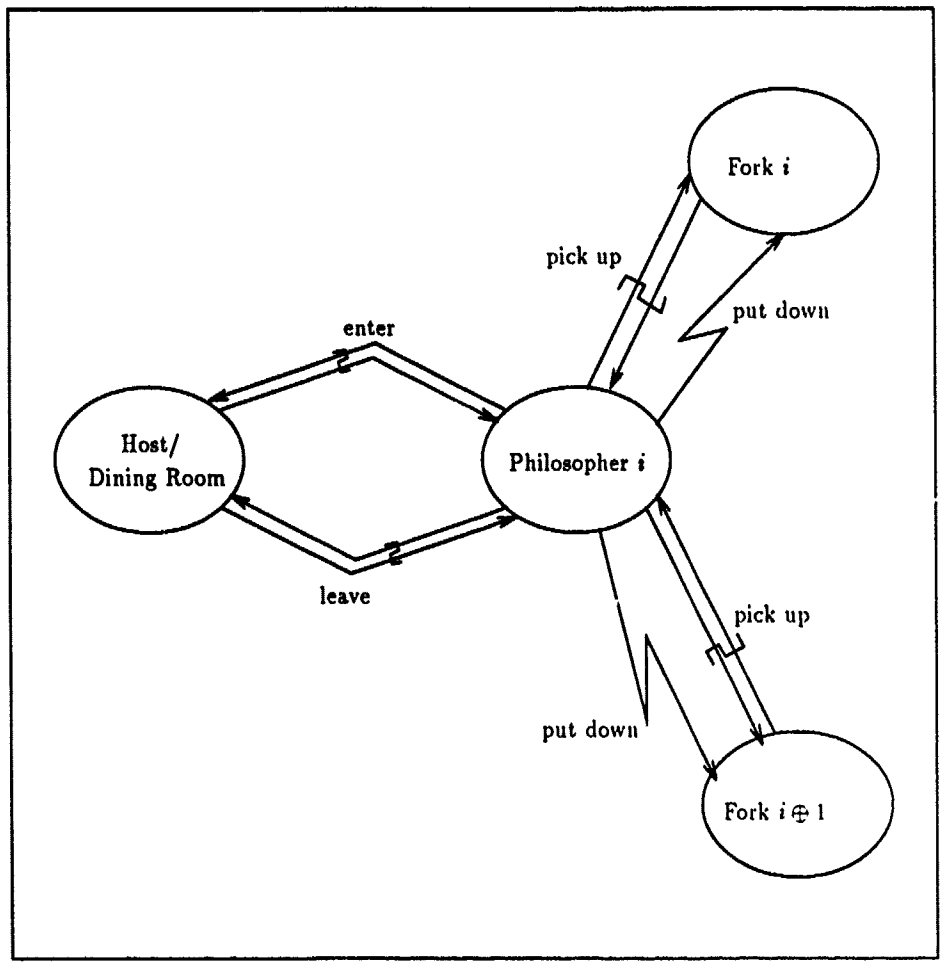


Figure 5.3. DARTS Design for Philosopher *i*

Before eating, each philosopher must ask the host's permission to enter the dining room by issuing the call ***Enter***. The host will only allow four philosophers into the dining room at a time so that deadlock will not occur.

After entering the dining room, the philosopher sits in his chair and tries to pick up the fork on his left (Fork i) and the fork on his right (Fork $i \oplus 1$). Table 5.1 shows the left and right fork numbers for each of the philosophers and Figure 5.4 shows the relative positions of the seats and the forks. (Note, the symbol $\oplus$ is used to denote modulo 5 addition (13:75).)

Table 5.1. Fork Numbering

Philosopher	Left Fork	Right Fork
0	0	1
1	1	2
2	2	3
3	3	4
4	4	0

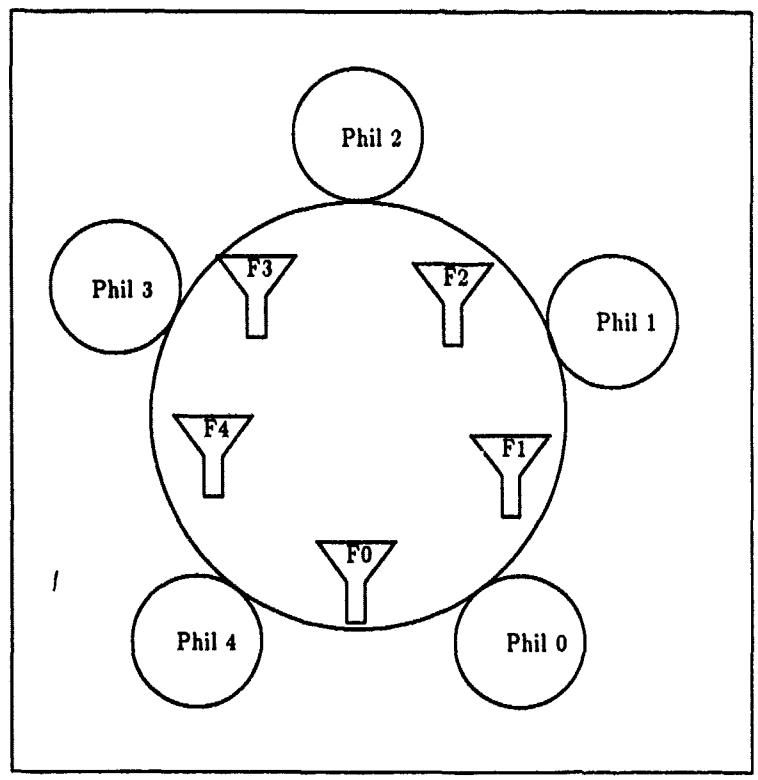


Figure 5.4. Fork Diagram for Dining Philosophers

After eating, the philosopher puts down both of his forks and notifies the host that he is leaving by issuing the call *Leave*. The calls *Enter* and *Leave* allow the host task to keep track of the number of philosophers currently in the dining room.

5.2.2 **Task Information.** There are eleven Ada tasks: one Host task; five Fork tasks; and five Philosopher tasks. The Host task has two Ada entry points: *enter* and *leave*. Each Fork task has two entries: ***pick up*** and ***put down***. The Philosopher tasks have no entries; each philosopher places calls to the Host and Forks. The twelve entries are enumerated in Table 5.2. Philosopher i repeats the eating–thinking cycle with frequency f_i.

The precedence requirements are defined by the order in which the entry calls can be made. The following list uses CSP notation to show the order of calls for Philosopher i:

$$\begin{aligned} Philosopher_i \;=\; & (i.enter \longrightarrow i.pick\ up\ fork\ i \longrightarrow i.pick\ up\ fork\ (i \oplus 1) \\ & \longrightarrow i.put\ down\ fork\ (i \oplus 1) \longrightarrow i.put\ down\ fork\ i \longrightarrow \\ & i.leave \longrightarrow Philosopher_i) \end{aligned}$$

The symbol $<\cdot$ is the precedence operator where $i <\cdot j$ means that i is dependent upon j. Every entry point is dependent upon all the other entry points in steady state because the problem is circular. The precedences shown here are partial precedences that only describe the previous precedence. The partial precedences for Philosopher i are shown below.

- enter $<\cdot$ leave
- pick up fork i $<\cdot$ enter
- put down fork i $<\cdot$ pick up fork i
- leave $<\cdot$ put down forks

In addition, each of the forks may have the following precedences, depending upon which philosopher is eating.

- pick up fork 1 $<\cdot$ pick up fork 0 (for Philosopher 0)

- pick up fork 2 $\ll$ pick up fork 1 (for Philosopher 1)
- pick up fork 3 $\ll$ pick up fork 2 (for Philosopher 2)
- pick up fork 4 $\ll$ pick up fork 3 (for Philosopher 3)
- pick up fork 0 $\ll$ pick up fork 4 (for Philosopher 4)
- put down fork 0 $\ll$ put down fork 1 (for Philosopher 0)
- put down fork 1 $\ll$ put down fork 2 (for Philosopher 1)
- put down fork 2 $\ll$ put down fork 3 (for Philosopher 2)
- put down fork 3 $\ll$ put down fork 4 (for Philosopher 3)
- put down fork 4 $\ll$ put down fork 0 (for Philosopher 4)

5.3 Application of the Ada Tasking Model

This section demonstrates how to apply the tasking model. The steps of the model are:

- Get Precedence Requirements;
- Create Entry Trace;
- Create Network of Entry Queues;
- Solve Network Equations;
- Gather Performance Statistics.

These steps are defined in Figure 4.1 in Chapter IV. Notice that "Model Arrival Patterns" was not included because the model assumes the arrivals are distributed exponentially.

5.3.1 **Get Precedence Requirements.** The entry points and their reference numbers are shown in Table 5.2. These numbers will be used as matrix indices throughout the remainder of the application of the model. Note that each Ada entry point is modeled as a separate queue.

Table 5.2. Entry Points

Entry #	Entry Name
1	Enter
2	Pick Up Fork 0
3	Pick Up Fork 1
4	Pick Up Fork 2
5	Pick Up Fork 3
6	Pick Up Fork 4
7	Put Down Fork 0
8	Put Down Fork 1
9	Put Down Fork 2
10	Put Down Fork 3
11	Put Down Fork 4
12	Leave

The procedure *Create Precedence Matrix* from Section B.7.1 in Appendix B was used to transfer the task information into a NxN matrix; N=12 in this case because there are 12 entry points. The precedence matrix, shown in Figure 5.5, contains Boolean values, such that T = True and F = False. (The False entries have been omitted from the figure in order to make it more readable.) The "T" in the matrix represents a precedence between the column and the row, i.e., $i \prec j$

	1	2	3	4	5	6	7	8	9	10	11	12
1		T	T	T	T	T	T	T	T	T	T	T
2			T				T					T
3				T				T				T
4					T				T			T
5						T				T		T
6		T									T	T
7											T	T
8							T					T
9								T				T
10									T			T
11										T		T
12	T											

Figure 5.5. Precedence Matrix

5.3.2 **Entry Trace.** This section is based upon the Dining Philosophers example in Hoare's text (13:75–81). There are three basic tasks in the implementation of the dining philosophers: host, philosophers, and forks; thus, there are three alphabets which are shown below with their respective behaviors.

5.3.2.1 **Philosophers.** Each Philosopher may enter or leave the dining room and pick up or put down forks. The alphabet for the philosophers is:

$$\begin{aligned} \alpha\ Philosopher_i \;=\; & \{i.enter, i.leave, i.pick\ up\ fork.i, i.pick\ up\ fork.(i \oplus 1), \\ & i.put\ down\ fork.i, i.put\ down\ fork.(i \oplus 1)\} \end{aligned}$$

A sample of Philosopher i's behavior is shown below.

$$\begin{aligned} Philosopher_i \;=\; & (i.enter \longrightarrow i.pick\ up\ fork.i \longrightarrow i.pick\ up\ fork.(i \oplus 1) \longrightarrow \\ & i.put\ down\ fork.(i \oplus 1) \longrightarrow i.put\ down\ fork.i \\ & \longrightarrow i.leave \longrightarrow Philosopher_i) \end{aligned}$$

5.3.2.2 **Forks.** Each Fork can be picked up or put down. The alphabet for the forks is:

$$\begin{aligned} \alpha\ Fork_i \;=\; & \{i.pick\ up\ fork.i, (i \ominus 1).pick\ up\ fork.i, \\ & i.put\ down\ fork.i, (i \ominus 1).put\ down\ fork.i\} \end{aligned}$$

The fork's behavior is:

$$\begin{aligned} Fork_i \;=\; & (i.pick\ up\ fork.i \longrightarrow i.put\ down\ fork.i \longrightarrow Fork_i \mid \\ & (i \ominus 1).pick\ up\ fork.i \longrightarrow (i \ominus 1).put\ down\ fork.i \longrightarrow Fork_i) \end{aligned}$$

5.3.2.3 **Host.** The host allows the philosophers to enter or leave the dining room. The alphabet for the host is:

α Host $= \bigcup_{i=0}^{4} \{i.enter,\ i.leave\}$

The host only allows philosophers to enter if the dining room is empty because there are no philosophers to leave.

$$\text{Host}_0 = (i.enter \longrightarrow Host_1)$$

The Host will allow the philosophers to enter or leave when there are 1 to 3 philosophers in the dining room.

$$\text{Host}_j = (\text{i.enter} \longrightarrow \text{Host}_{j+1} \mid \text{i.leave} \longrightarrow \text{Host}_{j-1}) \quad for\ j \in \{1,2,3\}$$

When four philosophers are in the dining room, the host will only allow philosophers to leave since four is the maximum number allowed in order to prevent deadlock.

$$\text{Host}_4 = (\text{i.leave} \longrightarrow \text{Host}_3)$$

5.3.2.4 **Concurrency.** The components work together concurrently and are described as follows:

Philosophers = $(Philosopher_0 \| Philosopher_1 \| Philosopher_2 \| Philosopher_3 \| Philosopher_4)$

Forks = $(Fork_0 \| Fork_1 \| Fork_2 \| Fork_3 \| Fork_4)$

Dining Philosophers = (Philosophers || Forks || Host)

5.3.2.5 **Trace.** A CSP-like trace can be created either manually or via automation. The trace for this implementation was created by embedding commands in the Ada program. Both the Ada code and entry trace are located in Appendix C. A portion of the trace is shown in Figure 5.6.

5.3.3 **Create Network of Entry Queues.** The next step in applying the Ada Tasking Model is to create the entry queue network. The queueing network may be drawn manually if the number of queues is small; otherwise, an NxN matrix is created, where the indices represent the entry queue numbers. Figure 5.7 shows the queueing network for the entry queues. Although this network only has sixteen queues, it is still unwieldy to draw; therefore, matrices are used for this model because they are much easier to manipulate and store on a computer.

Philosopher 0 enters dining room ⟶ Philosopher 0 picks up fork 0 ⟶
Philosopher 0 picks up fork 1 ⟶ Philosopher 1 enters dining room ⟶
Philosopher 2 enters dining room ⟶ Philosopher 3 enters dining room ⟶
Philosopher 2 picks up fork 2 ⟶ Philosopher 3 picks up fork 3 ⟶
Philosopher 3 picks up fork 4 ⟶ Philosopher 0 puts down fork 1 ⟶
Philosopher 1 picks up fork 1 ⟶ Philosopher 0 puts down fork 0 ⟶
Philosopher 0 leaves dining room ⟶ Philosopher 4 enters dining room ⟶
Philosopher 3 puts down fork 4 ⟶ Philosopher 4 picks up fork 4 ⟶
Philosopher 3 puts down fork 3 ⟶ Philosopher 2 picks up fork 3 ⟶
Philosopher 4 picks up fork 5 ⟶ Philosopher 3 leaves dining room ⟶
Philosopher 2 puts down fork 3 ⟶ Philosopher 4 puts down fork 5 ⟶
Philosopher 2 puts down fork 2 ⟶ Philosopher 1 picks up fork 2 ⟶
Philosopher 4 puts down fork 4 ⟶ Philosopher 2 leaves dining room ⟶
Philosopher 4 leaves dining room ⟶ ...

Figure 5.6. Entry Trace for Dining Philosophers

The r_{ji} matrix, shown in Figure 5.8, represents the probabilities of transitioning between queues, e.g., r_{ji} is the probability of moving from queue j to queue i. If r_{ji} equals zero, then the transition between the queues cannot occur. The queue interconnections are determined from the precedence matrix in Figure 5.5 and the r_{ji} values are calculated in Section 5.3.4.

Note that the matrix has been expanded from 12x12 to 16x16 matrix. The Leave queue will be used to model the thinking time for the philosophers; therefore, the Leave queue was expanded from one to five queues. This was done so that the queueing model could allow the philosophers to have their own "think" queue; thus, allowing the possibility for independent thinking rates.

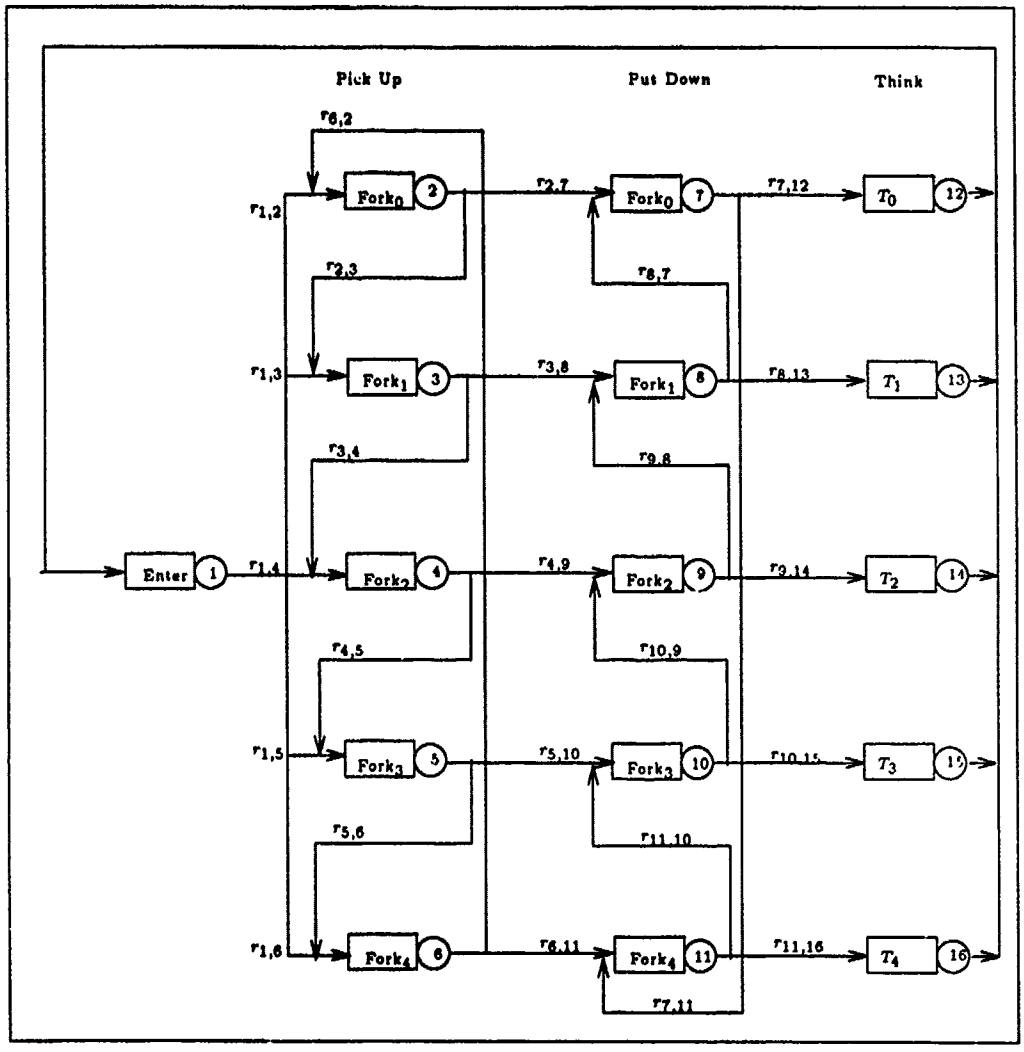


Figure 5.7. Queueing Network for Dining Philosophers

5.3.4 **Solve Network Equations.** This section calculates the r_{ji} probabilities for the general case; Section 5.6 solves the equations for a specific case. In order for the queueing network to be in steady state, no deaths will occur; therefore, the probability of leaving queues 12–16 and entering queue 1 is equal to 1.

$$r_{12,1} = r_{13,1} = r_{14,1} = r_{15,1} = r_{16,1} = 1$$

Note that this network is a closed system with five customers and that the external arrivals (γ_i) are 0 because the system is closed. The system has sixteen queues; therefore, there are sixteen simultaneous equations to solve of the form:

$$\lambda_i = \gamma_i + \sum_{j=1}^{N} r_{ji}\lambda_j \; for \; i = 1, 2, ..., 16 \tag{5.1}$$

Because the γ_i's are zero, this set of equations does not have a unique solution. However, λ_1 is equal to the sum of all the philosopher's arrival rates and can be substituted into the following set of equations to gain a unique solution.

$$\lambda_1 = r_{12,1}\lambda_{12} + r_{13,1}\lambda_{13} + r_{14,1}\lambda_{14} + r_{15,1}\lambda_{15} + r_{16,1}\lambda_{16}$$
$$\lambda_2 = r_{1,2}\lambda_1 + r_{6,2}\lambda_6$$
$$\lambda_3 = r_{1,3}\lambda_1 + r_{2,3}\lambda_2$$
$$\lambda_4 = r_{1,4}\lambda_1 + r_{3,4}\lambda_3$$
$$\lambda_5 = r_{1,5}\lambda_1 + r_{4,5}\lambda_4$$
$$\lambda_6 = r_{1,6}\lambda_1 + r_{5,6}\lambda_5$$
$$\lambda_7 = r_{2,7}\lambda_2 + r_{8,7}\lambda_8$$
$$\lambda_8 = r_{3,8}\lambda_3 + r_{9,8}\lambda_9$$
$$\lambda_9 = r_{4,9}\lambda_4 + r_{10,9}\lambda_{10}$$
$$\lambda_{10} = r_{5,10}\lambda_5 + r_{11,10}\lambda_{11}$$
$$\lambda_{11} = r_{6,11}\lambda_6 + r_{7,11}\lambda_7$$
$$\lambda_{12} = r_{7,12}\lambda_7$$
$$\lambda_{13} = r_{8,13}\lambda_8$$
$$\lambda_{14} = r_{9,14}\lambda_9$$
$$\lambda_{15} = r_{10,15}\lambda_{10}$$
$$\lambda_{16} = r_{11,16}\lambda_{11}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	$r_{1,2}$	$r_{1,3}$	$r_{1,4}$	$r_{1,5}$	$r_{1,6}$	0	0	0	0	0	0	0	0	0	0
2	0	0	$r_{2,3}$	0	0	0	$r_{2,7}$	0	0	0	0	0	0	0	0	0
3	0	0	0	$r_{3,4}$	0	0	0	$r_{3,8}$	0	0	0	0	0	0	0	0
4	0	0	0	0	$r_{4,5}$	0	0	0	$r_{4,9}$	0	0	0	0	0	0	0
5	0	0	0	0	0	$r_{5,6}$	0	0	0	$r_{5,10}$	0	0	0	0	0	0
6	0	$r_{6,2}$	0	0	0	0	0	0	0	0	$r_{6,11}$	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	$r_{7,11}$	$r_{7,12}$	0	0	0	0
8	0	0	0	0	0	0	$r_{8,7}$	0	0	0	0	0	$r_{8,13}$	0	0	0
9	0	0	0	0	0	0	0	$r_{9,8}$	0	0	0	0	0	$r_{9,14}$	0	0
10	0	0	0	0	0	0	0	0	$r_{10,9}$	0	0	0	0	0	$r_{10,15}$	0
11	0	0	0	0	0	0	0	0	0	$r_{11,10}$	0	0	0	0	0	$r_{11,16}$
12	$r_{12,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	$r_{13,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	$r_{14,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	$r_{15,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	$r_{16,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.8. r_{ji} Probability Transition Matrix

5.3.5 **Calculating the r_{ji}'s.** The r_{ji}'s denote probabilities of transitioning between queue j and i and the sum of the probabilities leaving a queue is 1. Note that each of the rows of the r_{ji} matrix in Figure 5.8 denotes the probability of leaving a specific queue, hence, the r_{ji}'s in each row must sum to 1. Thus, the following equations may be obtained from the r_{ji} matrix.

$r_{1,2} + r_{1,3} + r_{1,4} + r_{1,5} + r_{1,6} = 1$	probability of leaving queue 1
$r_{2,3} + r_{2,7} = 1$	probability of leaving queue 2
$r_{3,4} + r_{3,8} = 1$	probability of leaving queue 3
$r_{4,5} + r_{4,9} = 1$	probability of leaving queue 4
$r_{5,6} + r_{5,10} = 1$	probability of leaving queue 5
$r_{6,2} + r_{6,11} = 1$	probability of leaving queue 6
$r_{7,11} + r_{7,12} = 1$	probability of leaving queue 7
$r_{8,7} + r_{8,13} = 1$	probability of leaving queue 8
$r_{9,8} + r_{9,14} = 1$	probability of leaving queue 9
$r_{10,9} + r_{10,15} = 1$	probability of leaving queue 10
$r_{11,10} + r_{11,16} = 1$	probability of leaving queue 11

Each of the philosophers completes an eating–thinking cycle with frequency f_i. Thus, the arrival rate at queue 1 (λ_1) is equal to the sum of these frequencies, i.e., $\lambda_1 = \sum_{i=0}^{4} f_i$. The relationship between the frequencies is arbitrary, so assume that the frequency at which the philosophers eat is related by the following equation:

$$f_0 = \frac{f_1}{x_1} = \frac{f_2}{x_2} = \frac{f_3}{x_3} = \frac{f_4}{x_4} \tag{5.2}$$

where the x_i's are arbitrary constants. This relationship allows a general solution to be developed for each of the r_{ji} probabilities.

5.3.5.1 **Probabilities of Leaving Queue 1.** The following equation represents the sum for all the probabilities of leaving queue 1.

$$r_{1,2} + r_{1,3} + r_{1,4} + r_{1,5} + r_{1,6} = 1 \tag{5.3}$$

In order to solve for the r_{ji}'s, it is necessary to understand that each of these r_{ji}'s represent the probability of a philosopher selecting his left fork. For example, $r_{1,2}$ represents the probability that Philosopher 0 leaves queue 1 and picks up Fork 0.

Each of these probabilities are mutually exclusive and depend upon the rate at which the philosophers complete their eating-thinking cycle. The probability that Philosopher 0 leaves queue 1 is equal to his eating-thinking frequency (f_1) divided by the sum of all the eating-thinking frequencies (λ_1). Therefore, Philosopher 0 leaves queue 1 with the probability of:

$$r_{1,2} = \frac{f_0}{\lambda_1} = \frac{f_0}{\sum_{i=0}^{4} f_i} = \frac{f_0}{f_0 + f_1 + f_2 + f_3 + f_4} \tag{5.4}$$

This fraction can be simplified by exploiting the relationship between the eating-thinking frequencies.

$$f_0 = \frac{f_1}{x_1} = \frac{f_2}{x_2} = \frac{f_3}{x_3} = \frac{f_4}{x_4}$$

Thus,

$$f_1 = x_1 f_0$$

$$f_2 = x_2 f_0$$

$$f_3 = x_3 f_0$$

$$f_4 = x_4 f_0$$

Substituting these values into Eq 5.4 yields the following equation:

$$r_{1,2} = \frac{f_0}{(f_0 + x_1 f_0 + x_2 f_0 + x_3 f_0 + x_4 f_0)} = \frac{f_0}{f_0(1 + x_1 + x_2 + x_3 + x_4)} = \frac{1}{1 + x_1 + x_2 + x_3 + x_4}$$

Likewise, the probabilities that Philosophers 1-4 leave queue 1 and pick up their left forks are:

$$r_{1,3} = \frac{f_1}{\sum_{i=0}^{4} f_i} = \frac{x_1}{1 + x_1 + x_2 + x_3 + x_4}$$

$$r_{1,4} = \frac{f_2}{\sum_{i=0}^{4} f_i} = \frac{x_2}{1 + x_1 + x_2 + x_3 + x_4}$$

$$r_{1,5} = \frac{f_3}{\sum_{i=0}^{4} f_i} = \frac{x_3}{1 + x_1 + x_2 + x_3 + x_4}$$

$$r_{1,6} = \frac{f_4}{\sum_{i=0}^{4} f_i} = \frac{x_4}{1 + x_1 + x_2 + x_3 + x_4}$$

5.3.5.2 **Probabilities of Leaving Queue 2.** Queue 2 represents the entry *Pick Up Fork 0* and is called by either Philosopher 0 or Philosopher 4. Philosopher 0 leaves queue 2 and enters queue 3 with frequency f_0. Philosopher 4 leaves queue 2 and enters queue 7 with frequency f_4. Therefore,

$$r_{2,3} = \frac{f_0}{f_0+f_4} = \frac{1}{1+x_4}$$

$$r_{2,7} = \frac{f_4}{f_0+f_4} = \frac{x_4}{1+x_4}$$

5.3.5.3 **Probabilities of Leaving Queues 3–6.** The queues 3–6 also represent picking up a fork and the probabilities are derived similarly to those above. Therefore,

$$r_{3,4} = \frac{f_1}{f_0+f_1} = \frac{x_1}{1+x_1} \qquad r_{3,8} = \frac{f_0}{f_0+f_1} = \frac{1}{1+x_1}$$

$$r_{4,5} = \frac{f_2}{f_1+f_2} = \frac{x_2}{x_1+x_2} \qquad r_{4,9} = \frac{f_1}{f_1+f_2} = \frac{x_1}{x_1+x_2}$$

$$r_{5,6} = \frac{f_3}{f_2+f_3} = \frac{x_3}{x_2+x_3} \qquad r_{5,10} = \frac{f_2}{f_2+f_3} = \frac{x_2}{x_2+x_3}$$

$$r_{6,2} = \frac{f_4}{f_3+f_4} = \frac{x_4}{x_3+x_4} \qquad r_{6,11} = \frac{f_3}{f_3+f_4} = \frac{x_3}{x_3+x_4}$$

5.3.5.4 **Probabilities of Leaving Queues 7–11.** Queues 7–11 represent putting down forks. The following equations represent the probabilities of leaving these queues.

$$r_{7,12} = \frac{f_0}{f_0+f_4} = \frac{1}{1+x_4} \qquad r_{7,11} = \frac{f_4}{f_0+f_4} = \frac{x_4}{1+x_4}$$

$$r_{8,13} = \frac{f_1}{f_0+f_1} = \frac{x_1}{1+x_1} \qquad r_{8,7} = \frac{f_0}{f_0+f_1} = \frac{1}{1+x_1}$$

$$r_{9,14} = \frac{f_2}{f_1+f_2} = \frac{x_2}{x_1+x_2} \qquad r_{9,8} = \frac{f_1}{f_1+f_2} = \frac{x_1}{x_1+x_2}$$

$$r_{10,15} = \frac{f_3}{f_2+f_3} = \frac{x_3}{x_2+x_3} \qquad r_{10,9} = \frac{f_2}{f_2+f_3} = \frac{x_2}{x_2+x_3}$$

$$r_{11,16} = \frac{f_4}{f_3+f_4} = \frac{x_4}{x_3+x_4} \qquad r_{11,10} = \frac{f_3}{f_3+f_4} = \frac{x_3}{x_3+x_4}$$

5.4 SLAM II Simulation

Now that the r_{ji} probabilities have been calculated, it is possible to create a queueing model simulation in SLAM II (Simulation Language for Alternative Modeling) which is a FORTRAN-based simulation language (20). The SLAM II model has sixteen queues, where each queue represents one Ada entry point. (See Table 5.2.) One important difference between SLAM II and general queueing theory is that SLAM II uses times, whereas, queueing theory uses rates. However, the rate is simply the inverse of the time period.

The realistic service time for picking up and putting down forks can be approximated by zero. The time of interest for the forks is not the service time, but rather the wait time when the fork is in use by another philosopher. Thus, instead of modeling the service times as zero, the time will be used as part of the delay due to eating.

There are four steps necessary for the eating process:

- pick up left fork;
- pick up right fork;
- put down right fork; and
- put down left fork.

The eating service time will be divided between the pick up left fork and pick up right fork queues, such that the sum of the service times equals the total time spent eating. This allows a more realistic representation where the philosophers may be required to wait to pick up their forks. The queues for putting down the forks will have zero service times because the philosophers do not have to wait in line to put down the forks.

The queue where the philosopher notifies the host that he is leaving the dining room is used to model the thinking time. Each philosopher has his own thinking queue so that it is possible for all the philosophers to think concurrently and at differing rates.

The code and simulation output are contained in Appendix C. The statistics from the SLAM II simulations are contained within Section 5.6. The next section describes the Ada program for the Dining Philosophers.

5.5 Ada Implementation

The Ada implementation contains an array of five fork tasks, an array of five philosopher tasks, one host task, and two tasks used to collect statistics while the Ada program is executing. The collection tasks are used because I/O on a VERDIX system is a subprocess of the task and will suspend the task, thus, distorting the runtime statistics. The Ada code and entry trace are located in Appendix C. The statistics gathered from the Ada program and from the SLAM II simulation are located in the next section.

5.6 Dining Philosopher Solution

5.6.1 **Solve Network Equations.** The philosopher eating frequencies are related by the following equation:

$$f_0 = \frac{f_1}{x_1} = \frac{f_2}{x_2} = \frac{f_3}{x_3} = \frac{f_4}{x_4} \tag{5.5}$$

where the x_i's are arbitrary scaling constants. The sum of all the frequencies is equal to the arrival rate at Queue 1 in the queueing network:

$$\lambda_1 = \sum_{i=0}^{4} f_i \tag{5.6}$$

The solution presented here assumes that all the philosophers think at different rates, such that,

$x_1 = 2$

$x_2 = 3$

$x_3 = 4$

$x_4 = 5$

Each philosopher eats for approximately 1 hour. Let Philosopher 0 think for 9 hours and eat for 1 hour; therefore, his eating–thinking cycle takes approximately 10 hours and repeats with frequency $f_0 = \frac{1}{10}$. Plugging this value and the above x_i values into Eq 5.5 yields the following results. (The hour time unit is actually modeled as seconds.)

$f_0 = \frac{1}{10}$

$f_1 = \frac{1}{5}$

$f_2 = \frac{3}{10}$

$f_3 = \frac{2}{5}$

$f_4 = \frac{1}{2}$

and

$\lambda_1 = f_0 + f_1 + f_2 + f_3 + f_4 = \frac{3}{2}$

The r_{ji} probabilities shown below were calculated using MACSYMA. The batch file is contained in Appendix C.

$$r_{1,2} = \frac{1}{15}$$
$$r_{1,3} = \frac{2}{15}$$
$$r_{1,4} = \frac{1}{5}$$
$$r_{1,5} = \frac{4}{15}$$
$$r_{1,6} = r_{3,8} = r_{8,7} = \frac{1}{3}$$
$$r_{3,4} = r_{8,13} = \frac{2}{3}$$
$$r_{2,3} = r_{7,12} = \frac{1}{6}$$
$$r_{2,7} = r_{7,11} = \frac{5}{6}$$
$$r_{4,5} = r_{9,14} = \frac{3}{5}$$
$$r_{4,9} = r_{9,8} = \frac{2}{5}$$
$$r_{5,6} = r_{10,15} = \frac{4}{7}$$
$$r_{5,10} = r_{10,9} = \frac{3}{7}$$
$$r_{6,2} = r_{11,16} = \frac{5}{9}$$
$$r_{6,11} = r_{11,10} = \frac{4}{9}$$
$$r_{12,1} = r_{13,1} = r_{14,1} = r_{15,1} = r_{16,1} = 1$$

The sixteen simultaneous equations shown below are solved by substituting in the r_{ji} values and λ_1.

$$\lambda_1 = r_{12,1}\lambda_{12} + r_{13,1}\lambda_{13} + r_{14,1}\lambda_{14} + r_{15,1}\lambda_{15} + r_{16,1}\lambda_{16}$$
$$\lambda_2 = r_{1,2}\lambda_1 + r_{6,2}\lambda_6$$
$$\lambda_3 = r_{1,3}\lambda_1 + r_{2,3}\lambda_2$$
$$\lambda_4 = r_{1,4}\lambda_1 + r_{3,4}\lambda_3$$
$$\lambda_5 = r_{1,5}\lambda_1 + r_{4,5}\lambda_4$$
$$\lambda_6 = r_{1,6}\lambda_1 + r_{5,6}\lambda_5$$
$$\lambda_7 = r_{2,7}\lambda_2 + r_{8,7}\lambda_8$$
$$\lambda_8 = r_{3,8}\lambda_3 + r_{9,8}\lambda_9$$
$$\lambda_9 = r_{4,9}\lambda_4 + r_{10,9}\lambda_{10}$$
$$\lambda_{10} = r_{5,10}\lambda_5 + r_{11,10}\lambda_{11}$$
$$\lambda_{11} = r_{6,11}\lambda_6 + r_{7,11}\lambda_7$$
$$\lambda_{12} = r_{7,12}\lambda_7$$
$$\lambda_{13} = r_{8,13}\lambda_8$$

$\lambda_{14} = r_{9,14}\lambda_9$

$\lambda_{15} = r_{10,15}\lambda_{10}$

$\lambda_{16} = r_{11,16}\lambda_{11}$

The resulting λ values are:

$\lambda_1 = \frac{3}{2}$

$\lambda_2 = \lambda_7 = \frac{3}{5}$

$\lambda_3 = \lambda_8 = \frac{3}{10}$

$\lambda_4 = \lambda_9 = \frac{1}{2}$

$\lambda_5 = \lambda_{10} = \frac{7}{10}$

$\lambda_6 = \lambda_{11} = \frac{9}{10}$

$\lambda_{12} = \frac{1}{10}$

$\lambda_{13} = \frac{1}{5}$

$\lambda_{14} = \frac{3}{10}$

$\lambda_{15} = \frac{2}{5}$

$\lambda_{16} = \frac{1}{2}$

5.6.2 **Gather Performance Statistics.** The statistics of interest for the Dining Philosophers are the arrival rate (λ), service rate (μ), queue utilization (ρ), and service time (S). Note that the simulated values are often less than the theoretical values because the equations used to calculate the theoretical values are based upon the assumption that there is an infinite population when, in fact, there are only five entities circulating throughout the queueing network. The theoretical λ's, μ's, ρ's, and S's are shown in Table 5.3.

The utilization factor, ρ, is the "fraction of time that a server is busy" (16:19), i.e., the "ratio of the rate at which 'work' enters the system to the maximum rate (capacity) at which the system can perform this work" (16:18). Symbolically,

$$\rho = \frac{\lambda}{m\mu} \tag{5.7}$$

Table 5.3. Expected Queueing Statistics

Queue	λ	μ	ρ	S
1	1.5	2.000	0.75	0.5
2	0.6	1.000	0.60	1.0
3	0.3	1.000	0.30	1.0
4	0.5	1.000	0.50	1.0
5	0.7	1.000	0.70	1.0
6	0.9	1.000	0.90	1.0
7	0.6	—	—	0.0
8	0.3	—	—	0.0
9	0.5	—	—	0.0
10	0.7	—	—	0.0
11	0.9	—	—	0.0
12	0.1	0.111	0.90	9.0
13	0.2	0.250	0.80	4.0
14	0.3	0.429	0.70	2.3
15	0.4	0.667	0.60	1.5
16	0.5	1.000	0.50	1.0

where m is the number of available servers, λ is the queue arrival rate, and μ is the queue service rate. All the queues are single server queues; therefore, $m = 1$. Queues 7–11, put down forks, have zero service times and will not be included in this statistical analysis. Table 5.4 shows the simulated and theoretical utilization factors for the queues.

The service rates for the queues are used to approximate the eating and thinking times. Each eat cycle has four steps (pick up left fork, pick up right fork, put down right fork, and put down left fork) and the pick up left and right fork queues are used to model the eating time. Each pick up fork queue will have a service time equal to the eating time in order to simulate the wait time for a philosopher if the fork is in use. In the worst case, a philosopher will have to wait for both forks.

There are five thinking queues in order to allow the five philosophers to think concurrently. Tables 5.5 and 5.6 contain the average eating and thinking service times for the SLAM II simulations and the Ada implementation. Table 5.7 contains the eating-thinking cycle times.

Table 5.4. Utilization Factors

Queue	Expected Value	SLAM II Average	% error
1	0.75	0.2265	69.8
2	0.60	0.0939	84.4
3	0.30	0.0945	68.5
4	0.50	0.0922	81.2
5	0.70	0.0916	86.9
6	0.90	0.0934	89.6
7	0.00	0.0000	0.0
8	0.00	0.0000	0.0
9	0.00	0.0000	0.0
10	0.00	0.0000	0.0
11	0.00	0.0000	0.0
12	0.90	0.8461	6.0
13	0.80	0.3782	52.7
14	0.70	0.2152	69.3
15	0.60	0.1371	77.1
16	0.50	0.0929	81.4

Table 5.5. Thinking Service Times

	Expected Value	SLAM II Average	% error	Ada Average	% Error
Phil 0	9.000	9.139	1.52	9.486	5.12
Phil 1	4.000	4.085	2.10	4.346	7.96
Phil 2	2.333	2.325	0.36	2.391	2.41
Phil 3	1.500	1.481	1.27	1.571	4.52
Phil 4	1.000	1.003	0.30	0.933	6.70

Table 5.6. Eating Service Times

	Expected Value	SLAM II Average	% error	Ada Average	% Error
Phil 0	1.000	1.051	5.1	1.080	8.0
Phil 1	1.000	1.043	4.3	1.101	10.1
Phil 2	1.000	1.051	5.1	1.069	6.9
Phil 3	1.000	1.094	9.4	1.044	4.4
Phil 4	1.000	1.099	9.9	1.009	0.9

Table 5.7. Average Eating–Thinking Cycle

	Expected	SLAM II	%	Ada	%
	Value	Average	error	Average	Error
Phil 0	10.0	10.190	1.86	10.566	5.36
Phil 1	5.0	5.128	2.50	5.447	8.21
Phil 2	3.3	3.375	1.24	3.460	3.66
Phil 3	2.5	2.574	2.87	2.615	4.40
Phil 4	2.0	2.103	4.90	1.942	2.90

5.7 Discussion of Results

This section presents an overview of the results gained from applying the Ada Tasking Model, SLAM II model, and the Ada implementation.

The simulated queue utilization values did not match the expected values because there were only 5 entities circulating within the network. Finite population queueing networks are "self-regulating," meaning that "when the system gets busy, with many of these customers in the queue, then the rate at which additional customers arrive is in fact reduced, thus lowering the further congestion of the system" (16:106). Thus, the utilization was much lower than expected.

The service times are independent of the number circulating within the network; thus, the values for these statistics are accurate to within 10%.

In addition, the entry trace demonstrated that the addition of the Host to the Dining Philosophers Problem does, in fact, prevent deadlock. Thus, it was shown that the DARTS design is effective in preventing deadlock without requiring that the actual code be implemented.

This chapter has shown how to apply the Ada Tasking Model. As already mentioned, the simulated results are often less than the theoretical results because the theory is based upon an infinite population. It is not possible to exactly model an Ada program in SLAM II; however, the results from the simulation may be used to build confidence in the theoretical values derived for the queueing network.

Although this validation does not formally prove that the Ada Tasking Model is correct, it does demonstrate that formally modeling Ada tasking is feasible. In order for this model to be useful, it must be automated. The automation will make the application of the model much easier and allow changes to be made without causing the entire process to be repeated. A discussion on improving the Ada Tasking Model is presented in the next chapter.

VI. Conclusions and Recommendations

6.1 Motivation for Research

As software system requirements become more complex, software engineers must carefully design their systems to ensure that the systems adequately meet all the requirements, both functional and non-functional. Because real-time systems have timing constraints, in addition to the more traditional behavioral constraints, a comprehensive software design analysis model is required which incorporates performance, timing, and behavioral constraints. Although the Ada language tasking constructs are compiler independent, Ada tasking is dependent on its runtime environment; therefore, a formal model of Ada tasking is important in order for system designers to make realistic decisions when modeling Mission Critical Computer Resources (MCCR) systems.

6.2 Conclusions

- **Feasibility.**

  The main focus of this research effort was to determine the feasibility of developing a parameterized, formal model of Ada tasking. This research showed that such a parameterized model could be developed by creating a mathematical model which incorporated real-time scheduling and queueing theory.

- **Exponential Distribution Assumption.**

  The model is based upon the assumption that arrival and service times are exponential for entry queues. This assumption allowed Ada entry points to be modeled as M/M/1 queues. The model needs to be broadened to include general arrival and service distributions.

- **Modularity.**

  The model was built modularly so that changes, such as to the distributions, could be easily incorporated. Another purpose behind the modularity was to allow the model to be parameterized so that the model could be tailored for specific software applications and runtime environments.

- **Usability.**

  The Ada Tasking Model needs to be automated in order to be usable. The automation would also allow changes to be made easily.

- **Design Analysis Environment.**

  The model can be used in the future to develop a design analysis environment for real-time software systems that require Ada as the target language. Thus, given a specification for such a system, the design analysis environment can be used to obtain the information needed to support Ada software design decisions.

6.3 Recommendations

- **General Distributions.**

  The Ada Tasking Model assumes that arrival and service times are exponential for entry queues; thus, allowing each Ada entry point to be modeled as an M/M/1 queue. If the Ada tasking facility were modified so that it included timing guarantees, then the model would need to consider general arrival and service distributions. The queues would be modeled as G/M/1 for general arrival distributions, M/G/1 for general service distributions, and G/G/1 for both general arrival and service distributions. Function A234, *Model Arrival Patterns*, will need to be redesigned in the event that arrival distributions can be other than exponential.

- **Automation.**

  Additionally, the model should ultimately be automated to facilitate its application. The software engineer should only need to input the task information to the Ada Tasking Model which would then manipulate the information and produce the entry trace and performance statistics.

- **SADT.**

  This research used SADT to model the Ada tasking model. The SUN work stations at AFIT contain a CASE tool called SAtool which automate the application of SADT.

This tool should be used in the future because of its automated data dictionary and consistency checking.

- **Further Research.**

  Model Design Performance is divided into two functions: *Define Task Performance Requirements* (Function A1); and *Ada Tasking Model* (Function A2). (See Figure 3.2.) This research effort concentrated on developing the *Ada Tasking Model.* Future research should develop *Define Task Performance Requirements* and its input control *non-functional requirements.*

Appendix A. Glossary of Acronyms

AFIT	Air Force Institute of Technology
ATM	Ada Tasking Model
CPU	Central Processor Unit
CSP	Communicating Sequential Processes
DARTS	Design Approach for Real–Time Systems
DoD	Department of Defense
ECS	Embedded Computer System
FCFS	First–Come–First–Serve
IOP	Input/Output Processor
LRM	Language Reference Manual
MCCR	Mission Critical Computer Resources
pdf	Probability Density Function
PDF	Probability Distribution Function
RTE	Runtime Environment
RR	Round Robin
RSTA	Real–Time Structured Analysis
SADT	Structured Analysis and Design Technique
SJF	Shortest–Job–First
UNITY	Unbounded Nondeterministic Iterative Transformations

Appendix B. *Detailed Design*

B.1 Environment Model

Figure B.1 represents the environmental model.

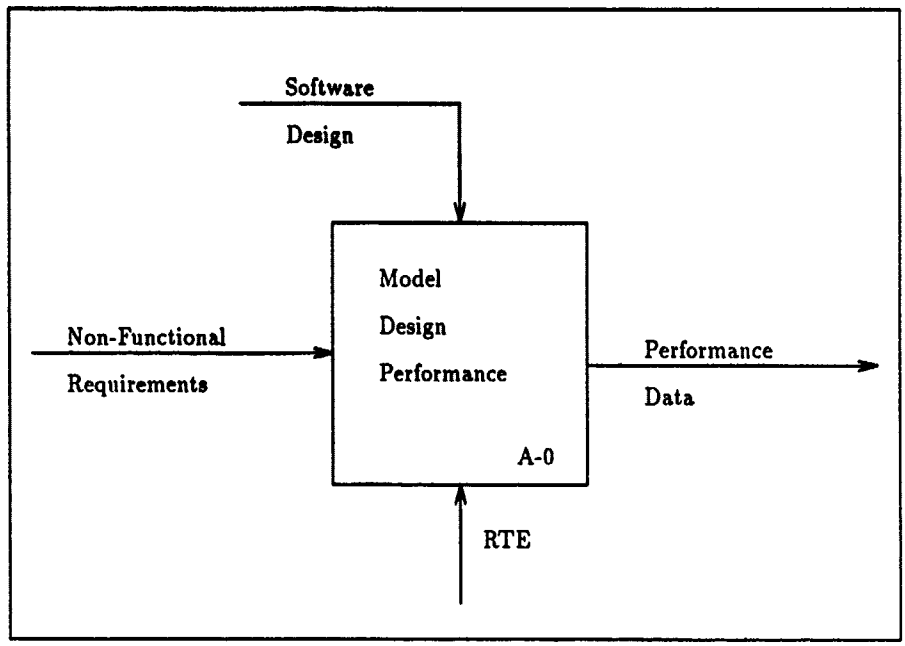


Figure B.1. Environment Model

B.2 Model Design Performance

Figure B.2 represents the A0 level of the model. This level is decomposed further to levels A1, *Define Task Performance Requirements*, and A2, *Ada Tasking Model.*

B.3 Define Task Performance Requirements

Level A1, labeled *Define Task Performance Requirements*, will be designed in a later research effort.

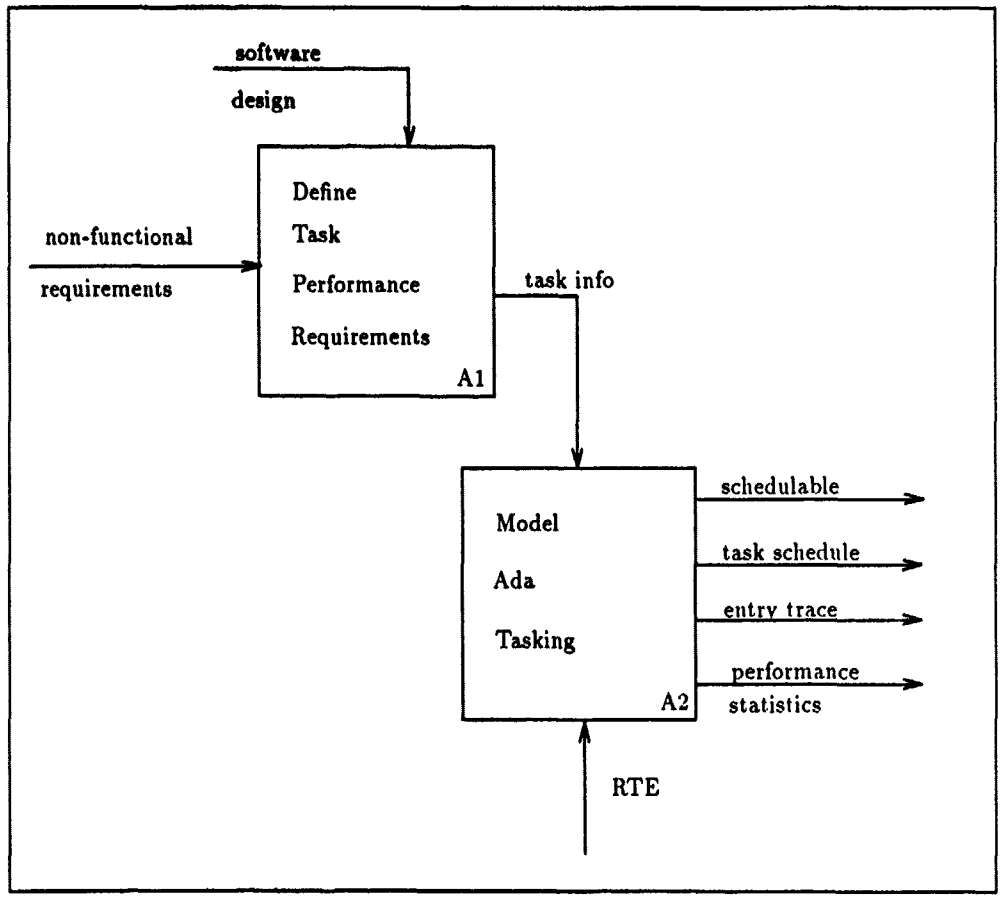


Figure B.2. Level A0

B.4 Ada Tasking Model

Level A2, labeled *Ada Tasking Model*, models the performance of a given Ada design. Function A2 (shown in Figure B.3) returns a Boolean flag, labeled schedulable; a task schedule; an entry trace; and performance statistics, based upon the information included in *task info* and *RTE*.

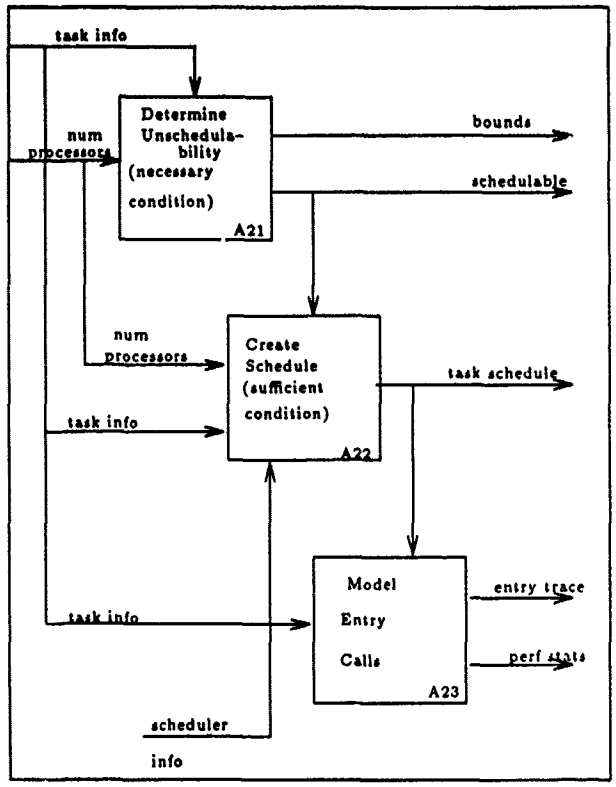


Figure B.3. Level A2

Level A2 is further decomposed to three functions: level A21, *Determine Unschedulability*; level A22, *Create Schedule*; and level A23, *Model Entry Calls*. These functions are described in detail in the following sections.

B.5 Determine Unschedulability

The steps to determine unschedulability are: Determine Pairwise Task Compatibility; Find Maximal Compatible Sets of Tasks; and Determine Lower Bound on Number of Processors. (See Figure B.4.)

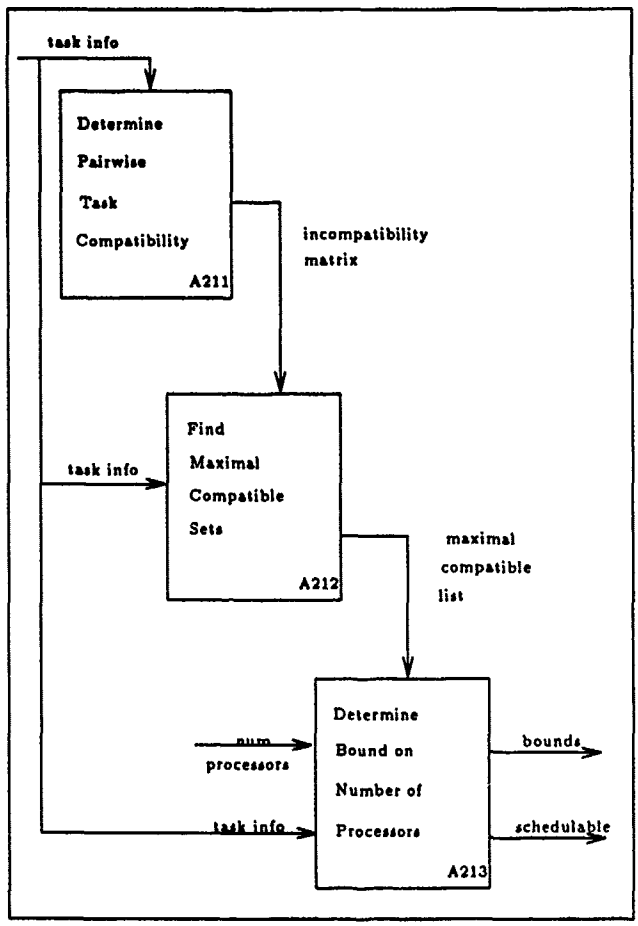


Figure B.4. Determine Unschedulability

B.5.1 **Determine Pairwise Task Compatibility.** Create an NxN incompatibility matrix where N is the number of tasks to be scheduled. As previously mentioned, two tasks are incompatible if they cannot be scheduled together on a single processor. The incompatibility occurs if the sum of their load factors is greater than 1 or if the sum of their

execution times is larger than the greatest common divisor of their periods (21:55). The following pseudocode describes the algorithm to create the incompatibility matrix (21:70).

```
procedure Create_Incompatibility_Matrix is
   i,j = indices of loop counters
   N   = number of tasks
   T   = array (1..N) of task periods
   E   = array (1..N) of task execution times
   M   = array (1..N, 1..N) of Boolean
       -- M is the incompatibility matrix
         -- True means incompatible
         -- False means not incompatible, i.e. compatible
   gcd is a separate function which returns the greatest common
      divisor of two integers

begin procedure
   for i in 1..N loop
      for j in 1..N loop
         if i=j then
            M(i,i) = False  -- every task is compatible
                            -- with itself
         else
            if Ei + Ej <= gcd (Ti, Tj) then
               -- task i & j are compatible
               M(i,j) = False
            else
               -- task i & j are incompatible
               M(i,j) = True
            end if
         end if
      end loop
   end loop
end procedure
```

B.5.2 **Find Maximal Compatible Sets of Tasks.** The functions, Zeroj, Zero1, list, push, and pop, used within the algorithm, are defined first before describing the algorithm itself. The call Zeroj (j, M) takes the matrix M and zeros out all the entries in the jth column and the jth row (21:77). The call Zero1 (j, M) takes the matrix M and zeros all the rows that have 1's in the jth column and then zeros out the entries in the jth row and column (21:77). Function list(j) returns the list of tasks represented by a 1 in row

j. The other functions, push and pop, refer to stack operations. Pushing an item places it on the top of the stack and popping removes an item from the top of the stack. The algorithm has three stacks: job stack, job complement stack, and array stack.

The following algorithm uses the incompatibility matrix created in the previous step to produce a list of maximal compatible sets which are sets of tasks which do not exclude each other from being scheduled on the same processor. Seward's Algorithm 2.1 (21:81–82) was found to be too general; the problems with this algorithm are:

1. It assumes that the tree will always have a left and right leaf. This is not true because most of the left branches branch out further, leaving only a right leaf. The algorithm always tries to get the maximal compatible list from the left leaf even when it does not exist.
2. Sometimes a right branch branches down another level, but the algorithm never considers this possibility. When this occurs, it is necessary to pop an extra array and job list from their stacks in order to back up to the next node in the tree.

The modified algorithm is shown below.

```
Step 1.   Find the maximum number of ones in any column.
          This task will become the root of the tree.

          decision:  if none of the columns contain any
          1's, then move on to Step 3, skipping Step 2.

Step 2.   push job (column number) onto job stack

          push list (j) onto job complement stack

          push Zeroj (j, M) onto array stack

          set incompatibility array R = Zeroj (j, M)

          Goto Step 4.

Step 3.   R now has only zeroes.

          handle the left leaf
```

```
          (the tree will only have left leaves on
     the first time through the algorithm and if a
     right branch branches down a whole level.)
          -- get maximal compatible (MC) list and store it

          backup to node and go down right side
          -- pop job stack twice
          -- push job complement list onto job stack

     handle the right leaf
          -- get maximal compatible (MC) list and store it

          backup 2 nodes
          -- pop job stack twice

          if the right branch was down a whole
          level (having two leaves) then have
          to pop an extra job list and array
          from the stacks)
               -- pop job stack
               -- pop array stack

     prepare to go down right branch
          -- pop job complement stack
          -- push job complement onto job stack
          -- pop array stack twice

     if back at the root then stop.
     else  R = Zero1 (last array popped)

Step 4.  (An extension of step 1.)

     Find the maximum number of ones in any column.

     decision:  if there are zeros in the matrix
     and this is not the first time through (depth
     first search) then the tree is branching into
     an extra level.

     goto Step 2.
```

B.5.9 **Determine Bound on Number of Processors.** An estimate of the lower bound can be found by summing the load factors and taking the ceiling of the sum. In other words, the minimum number of processors is equal to

$$\lceil (\sum_{i=1}^{N} \frac{E_i}{T_i}) \rceil \tag{B.1}$$

This lower bound offers the least number of processors that is possible. An approach which consistently produces a more realistic lower bound is described below.

Determining the minimal number of processors is part of the Set Covering Problem (21:207). Finding the minimal covering, however, is not enough because each of the job sets must be load consistent if they are to be scheduled on a single processor. A load consistent set is a set of tasks whose load factors sum is less than unity. Therefore, the minimal covering only gives a lower bound for the minimum number of processors required.

Because a valid schedule must have load consistent sets, every minimal covering must be found and every irredundant cover that is not minimal must also be found (21:211-212). Seward defines an irredundant covering as a covering in which the removal of one of the maximal compatible (MC) sets causes the set to no longer be a cover (21:211-212).

The following is an algorithm for finding the lower bound of processors required.

```
Step 1:  create a cover table where the rows represent the
         MC lists and the columns represent the jobs.

         Place a 1 in the table to represent that a job is
         present in the MC list.

Step 2:  if a column contains a single 1, then the corresponding
         MC list is an essential MC because the job is only
         contained in that MC list.  This MC must be contained
         in all the solutions.

         Remove the MC from the cover table and all the jobs
         which are contained in the MC list.

Step 3:  Check the table for equivalent rows and remove them.

Step 4:  Repeat Steps 2 & 3 until no more essential MC lists
```

```
        are found.

Step 5: Find the list of maximal compatible sets for the
        jobs which are not contained in the essential MC lists.

Step 6: Append the essential MCs to the new MC lists.  The
        number of lists gives the lower bound on the required
        number of processors.
```

B.6 Create Schedule

Function A22, *Create Schedule*, creates a schedule of the tasks based upon the task scheduler for the RTE. This model does not worry about finding the optimum schedule because this is often an NP-complete problem (21). For example, the seemingly simple case of finding the optimum solution for a system with a nonpreemptive scheduling algorithm, independent tasks, and unequal execution times is an NP-complete problem (21:12).

B.7 Model Entry Calls

Figure B.5 depicts level A23 of the model design. This level has six functions: *Get Entry Precedence Requirements; Create Entry Trace; Create Network of Entry Queues; Model Arrival Patterns; Solve Network Equations*; and *Gather Performance Statistics*. The following sections describe each of these functions.

B.7.1 **Get Entry Precedence Requirements.** The precedence information is received from the designer who inputs the task information. This information is transferred to an NxN matrix, where N is the number of entry points. The matrix contains Boolean values with dependencies denoted by TRUE. The column depends upon the row. If entry j depends upon entry i, then M(i,j) = TRUE; M(i,i) = FALSE because a task cannot depend upon itself.

The following pseudocode describes the algorithm to create the precedence matrix. The function *is_precedence(i,j)* checks the task information to see whether or not the entry *j* depends upon the entry *i*.

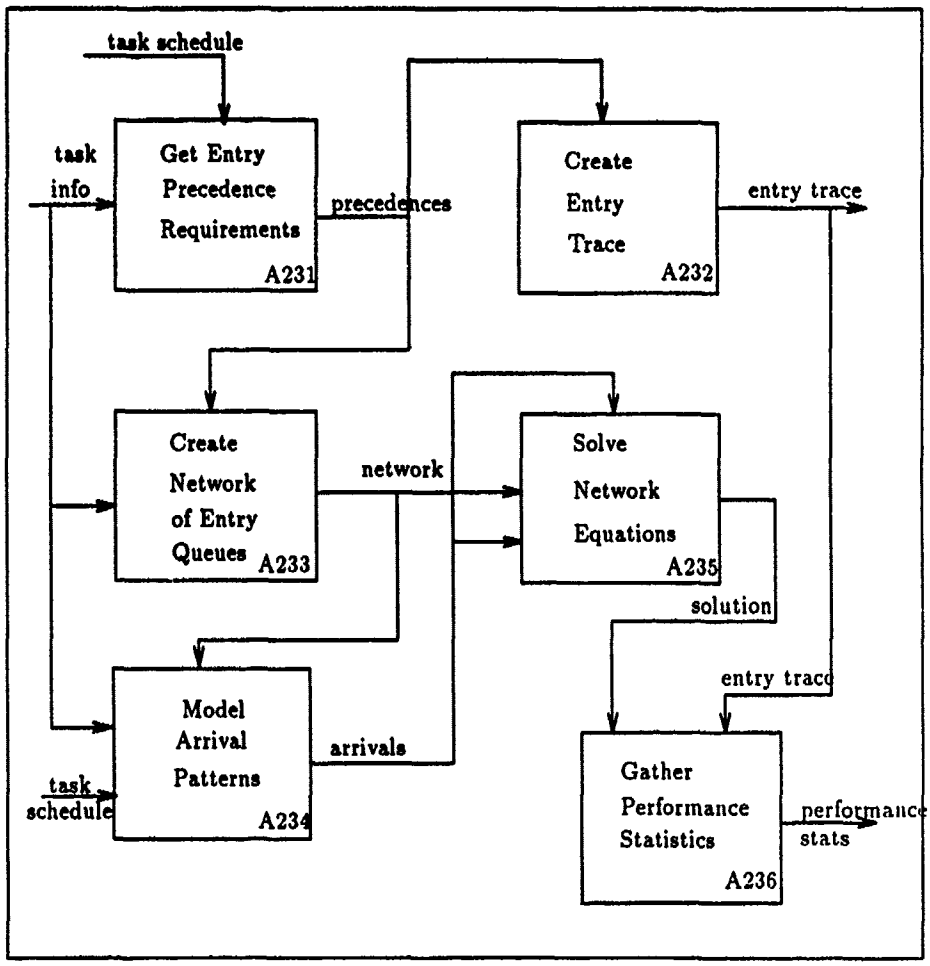


Figure B.5. Model Entry Calls – Level A23

```
procedure Create_Precedence_Matrix is
   i,j = indices of loop counters
   N   = number of entry points
   T   = array (1..N) of entry precedence constraints
   M   = array (1..N, 1..N) of Boolean
       -- M is the precedence matrix
         -- True means precedence constraint, i.e., dependent
         -- False means no constraint, i.e., independent
   is_precedence(j,i) is a separate function which returns TRUE
      if j is constrained by i.

begin procedure
   for i in 1..N loop
      for j in 1..N loop
         if i=j then
            M(i,i) = False  -- an entry does not depend
                            -- upon itself
         else
            if is_precedence(T(j), T(i)) then
               -- task j depends upon i
               M(i,j) = True
            else
               -- task i & j are independent
               M(i,j) = False
            end if
         end if
      end loop
   end loop
end procedure
```

B.7.2 **Create Entry Trace.** The entry trace will be created using a CSP-like language. The entry points are modeled as members of the alphabet. Hoare suggests that CSP can be represented by a LISP program (13:47). The validation model in this thesis incorporates the entry calls into an Ada program.

The entry points are modeled as members of an alphabet. Each task, or possibly subsystem, can have its own alphabet. If two tasks run concurrently and if an event is in both of the alphabets, then the event must occur simultaneously. If the event is in only one of the alphabets, then it may occur independently (13:68-69). The same interaction is true for "N" concurrent tasks.

CSP ignores timing details, but, if the entry trace is automated, then the addition of a time stamp can easily be added. These times will be useful for giving general timing relationships between different entry calls.

B.7.3 **Create Network of Entry Queues.** The queues represent the entry points, i.e., the accept statements. The arrival rate is determined by the calling task(s) and the queues will be represented by M/M/1 queues. The interconnections between the entry queues are determined by the precedence matrix developed in Section B.7.1.

This segment of the model may be described by manually drawing the queueing network if the number of queues is small. Otherwise, an NxN probability matrix is created, where the indices of the matrix represent entry queues. This matrix is similar to the precedence matrix described above except that the entries are values between 0 and 1 which denote the probability of leaving queue i (represented by row i) and entering queue j (represented by column j).

This matrix is referred to as the matrix of transition probabilities (P) for discrete systems and as the matrix of transition rates (Q) for continuous systems (17:53). This model will use the Q matrix because it assumes exponential distributions which imply that the system is continuous.

B.7.4 **Model Arrival Patterns.** The arrival distributions for the model developed in this thesis are assumed to be Poisson; therefore, the interarrival process is exponential. (The notation used in this section is consistent with that used by Kleinrock (16)).

The Poisson distribution is shown in the following equation.

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \; for \; k \geq 0, \; t \geq 0 \qquad \text{(B.2)}$$

λ is the average rate at which customers arrive at the queue. The average time between arrivals is $1/\lambda$. $P_k(t)$ is the probability of k arrivals during the time interval (0,t).

The time between arrivals is exponentially distributed; therefore, the interarrival times are exponential.

The exponential Probability Distribution Function (PDF) (16:65) for arrivals is denoted as:

$$A(t) = 1 - e^{-\lambda t} \quad for\ t \geq 0 \tag{B.3}$$

And the exponential Probability Density Function (pdf) (16:65) is:

$$a(t) = \frac{d\,A(t)}{dt} = \lambda e^{-\lambda t} \quad for\ t \geq 0 \tag{B.4}$$

The arrival rate (λ) for each of the queues is placed in a row vector, where the index refers to the queue number from Section B.7.1.

B.7.5 **Solve Network Equations.** Because the queues are assumed to be M/M/1, the network can be modeled using the Jackson network equation below (16:149-150),

$$\lambda_i = \gamma_i + \sum_{j=1}^{N} \lambda_j r_{ji} \quad for\ i = 1, 2, ..., N \tag{B.5}$$

where the r_{ji}'s are the rates described in the Q matrix in Section B.7.3 and the λ's are the arrival rates described in Section B.7.4.

There is an equation for each queue. The set of equations may be solved by hand or simultaneously by placing them in a matrix and using a mathematical software program. Figure B.6 shows one queue out of the system of queues.

External arrivals (γ_i) are arrivals from outside the network which have the Poisson distribution. Internal arrivals $\lambda_k r_{ki}$ arrive from other queues within the network. Feedback is represented by $\lambda_i r_{ii}$. External departures leave the system entirely and are represented as $\lambda_i(1 - \sum_j r_{ij})$. Internal departures leave Queue i as $\lambda_i r_{ij}$ and are internal arrivals at Queue j.

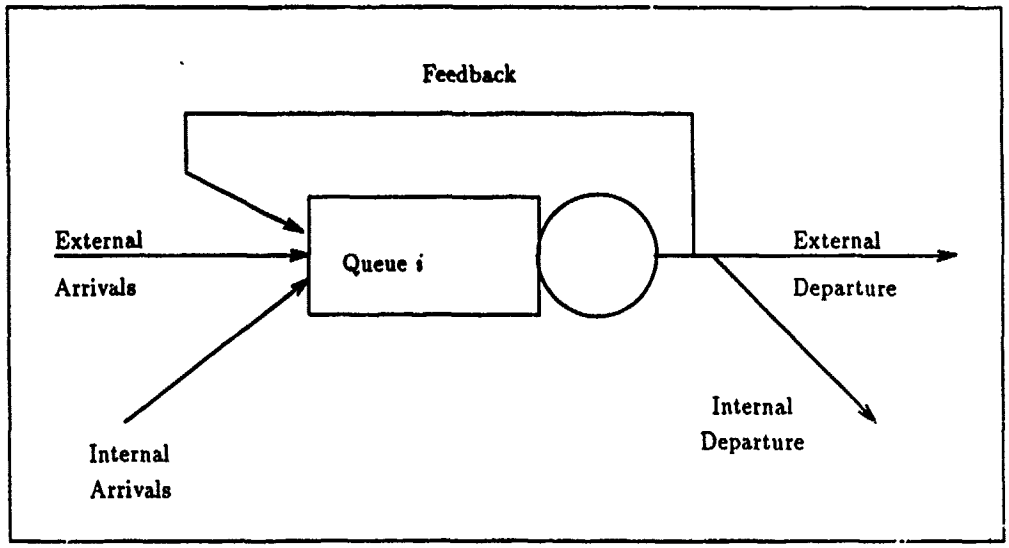


Figure B.6. Queue i in Network

B.7.6 **Gather Performance Statistics.** The entry trace generates a sequence of the entry calls. The remainder of the information gained from the entry trace depends upon the implementation of the trace. This model assumes that the trace will be implemented as output statements within the program and that the events will be time stamped. Thus, information gathered from the entry trace, in addition to the list of events, is the number of times each entry was called, when the calls were made, and general information for the interarrival and service distributions.

The statistics gathered from the queueing network are averages. The information may include:

- arrival rate (λ)
- service rate (μ)
- utilization of queues (ρ)
- time in queue (T)
- number in queue ($\bar{N}_q$)
- service time ($S(y)$)

- wait time ($W(y)$)

The values of λ and μ are calculated in the function *Solve Network Equations*. The equations for the remaining variables are shown below:

$$\rho = \lambda/\mu \tag{B.6}$$

$$T = \frac{1/\mu}{1-\rho} \tag{B.7}$$

$$\bar{N} = \lambda T = \frac{\rho}{1-\rho} \tag{B.8}$$

$$S(y) = 1 - e^{-\mu(1-\rho)y} \quad for\ y \geq 0 \tag{B.9}$$

$$W(y) = 1 - \rho e^{-\mu(1-\rho)y} \quad for\ y \geq 0 \tag{B.10}$$

B.8 Data Dictionary

B.8.1 List of Activities

```
A-0     MODEL DESIGN PERFORMANCE
A1      DEFINE TASK PERFORMANCE REQUIREMENTS
A2      Ada TASKING MODEL
A21     DETERMINE PAIRWISE TASK COMPATIBILITY
A22     CREATE SCHEDULE
A23     MODEL ENTRY CALLS
A231    GET ENTRY PRECEDENCE REQUIREMENTS
A232    CREATE ENTRY TRACE
A233    CREATE NETWORK OF ENTRY QUEUES
A234    MODEL ARRIVAL PATTERNS
A235    SOLVE NETWORK EQUATIONS
A236    GATHER PERFORMANCE STATISTICS
```

```
|
NAME:  Ada TASKING MODEL
TYPE:  ACTIVITY
PROJECT:  ATM
NUMBER:  A2
DESCRIPTION:
     The Ada tasking model models the performance of the Ada
     design which was developed by a method such as DARTS.
     The model returns a task schedule, and event trace, and
     performance statistics.
INPUTS:  none
OUTPUTS:  SCHEDULABLE
          TASK SCHEDULE
          ENTRY TRACE
          PERFORMANCE STATISTICS
CONTROLS:  TASK INFO
MECHANISMS:  RTE
PARENT ACTIVITY:  MODEL DESIGN PERFORMANCE
REFERENCE:  Appendix B
|
NAME:  CREATE ENTRY TRACE
TYPE:  ACTIVITY
PROJECT:  ATM
NUMBER:  A232
DESCRIPTION:
     This function creates an event trace which is a linear
     list of the entry calls.
INPUTS:  none
```

OUTPUTS: ENTRY TRACE
CONTROLS: PRECEDENCES
MECHANISMS: none
PARENT ACTIVITY: MODEL ENTRY CALLS
REFERENCE: Appendix B
|
NAME: CREATE NETWORK OF ENTRY QUEUES
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A233
DESCRIPTION:
The network of entry queues is developed from the entries which are input from the software design.
INPUTS: TASK INFO
OUTPUTS: NETWORK
CONTROLS: PRECEDENCES
MECHANISMS: none
PARENT ACTIVITY: MODEL ENTRY CALLS
REFERENCE: Appendix B
|
NAME: CREATE SCHEDULE
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A22
DESCRIPTION:
Create schedule creates a schedule based upon the task information, task scheduler, and number of processors available.
INPUTS:
NUM PROCESSORS
TASK INFO
OUTPUTS: TASK SCHEDULE
CONTROLS: SCHEDULABLE
MECHANISMS: SCHEDULER INFO
PARENT ACTIVITY: Ada TASKING MODEL
REFERENCE: Appendix B
|
NAME: DEFINE TASK PERFORMANCE REQUIREMENTS
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A1
DESCRIPTION:
Defining the task performance requirements will be designed in a follow-on thesis.
INPUTS: NON-FUNCTIONAL REQUIREMENTS

OUTPUTS: TASK INFO
CONTROLS: SOFTWARE DESIGN
MECHANISMS: none
PARENT ACTIVITY: MODEL DESIGN PERFORMANCE
REFERENCE: Appendix B
|
NAME: DETERMINE BOUND ON NUMBER OF PROCESSORS
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A213
DESCRIPTION:
This function estimates the upper and lower bounds of the required number of processors for the given set of tasks.
INPUTS:
TASK INFO
NUM PROCESSORS
OUTPUTS:
BOUNDS
SCHEDULABLE
CONTROLS: MAXIMAL COMPATIBLE LIST
MECHANISMS: none
PARENT ACTIVITY: DETERMINE UNSCHEDULABILITY
REFERENCE: Appendix B
|
NAME: DETERMINE PAIRWISE TASK COMPATIBILITY
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A211
DESCRIPTION:
This function compares every pair of tasks to see if they can co-exist on the same processor. An incompatibility occurs if the sum of their load factors is greater than unity or if the sum of their execution times is larger than the greatest common divisor of their periods.
INPUTS: none
OUTPUTS: INCOMPATIBILITY MATRIX
CONTROLS: TASK INFO
MECHANISMS: none
PARENT ACTIVITY: DETERMINE UNSCHEDULABILITY
REFERENCE: Appendix B
|
NAME: DETERMINE UNSCHEDULABILITY
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A21

DESCRIPTION:
Determine unschedulability is a necessary condition for determining if a set of tasks can be scheduled.
INPUTS: NUM PROCESSORS
OUTPUTS:
BOUNDS
SCHEDULABLE
CONTROLS: TASK INFO
MECHANISMS: none
PARENT ACTIVITY: Ada TASKING MODEL
REFERENCE: Appendix B
|
NAME: FIND MAXIMAL COMPATIBLE SETS
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A212
DESCRIPTION:
This function uses the incompatibility matrix to produce a list of maximal compatible sets which are sets of tasks which do not exclude each other from being scheduled on the same processor.
INPUTS: TASK INFO
OUTPUTS: MAXIMAL COMPATIBLE LIST
CONTROLS: INCOMPATIBILITY MATRIX
MECHANISMS: none
PARENT ACTIVITY: DETERMINE UNSCHEDULABILITY
REFERENCE: Appendix B
|
NAME: GATHER PERFORMANCE STATISTICS
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A236
DESCRIPTION:
This function generates statistics based upon the entry trace and the queueing network of entries.
INPUTS: none
OUTPUTS: PERFORMANCE STATISTICS
CONTROLS:
SOLUTION
ENTRY TRACE
MECHANISMS: none
PARENT ACTIVITY: MODEL ENTRY CALLS
REFERENCE: Appendix B
|
NAME: GET ENTRY PRECEDENCE REQUIREMENTS

TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A231
DESCRIPTION:
The precedence requirements are input by the designer and this function converts them into a format acceptable for the Ada Tasking Model.
INPUTS: TASK INFO
OUTPUTS: PRECEDENCES
CONTROLS: SCHEDULABLE
MECHANISMS: none
PARENT ACTIVITY: MODEL ENTRY CALLS (A23)
REFERENCE: Appendix B
|
NAME: MODEL ARRIVAL PATTERNS
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A234
DESCRIPTION:
This function describes the arrival distributions for each of the entry queues in the network developed in A233.
INPUTS:
TASK INFO
TASK SCHEDULE
OUTPUTS: ARRIVALS
CONTROLS: NETWORK
MECHANISMS: none
PARENT ACTIVITY: MODEL ENTRY CALLS (A23)
REFERENCE: Appendix B
|
NAME: MODEL DESIGN PERFORMANCE
TYPE: ACTIVITY
PROJECT: ATM
NUMBER: A-0
DESCRIPTION:
Modeling design performance is accomplished after the initial design has been accomplished by a method such as DARTS.
INPUTS: NON-FUNCTIONAL REQUIREMENTS
OUTPUTS: PERFORMANCE STATISTICS
CONTROLS: SOFTWARE DESIGN
MECHANISMS: RTE
PARENT ACTIVITY: none
REFERENCE: Appendix B

```
|
NAME:  MODEL ENTRY CALLS
TYPE:  ACTIVITY
PROJECT:  ATM
NUMBER:  A23
DESCRIPTION:
     Model entry calls models the entries in the Ada design
     and returns an entry trace and performance statistics.
INPUTS:  TASK INFO
OUTPUTS:
ENTRY TRACE
PERFORMANCE STATISTICS
CONTROLS:  TASK SCHEDULE
MECHANISMS:  none
PARENT ACTIVITY:  Ada TASKING MODEL
REFERENCE:  Appendix B
|
NAME:  SOLVE NETWORK EQUATIONS
TYPE:  ACTIVITY
PROJECT:  ATM
NUMBER:  A235
DESCRIPTION:
     This function solves the network of equations based upon
     the arrival distributions.
INPUTS:  NETWORK
OUTPUTS:  SOLUTION
CONTROLS:  ARRIVALS
MECHANISMS:  none
PARENT ACTIVITY:  MODEL ENTRY CALLS
REFERENCE:  Appendix B
```

B.8.2 List of Data Elements

```
ARRIVALS
BOUNDS
ENTRY TRACE
INCOMPATIBILITY MATRIX
MAXIMAL COMPATIBLE LIST
NETWORK
NON-FUNCTIONAL REQUIREMENTS
NUM PROCESSORS
PERFORMANCE DATA
PERFORMANCE STATISTICS
PRECEDENCES
```

RTE
SCHEDULABLE
SCHEDULER INFO
SOFTWARE DESIGN
SOLUTION
TASK INFO
TASK SCHEDULE

|
NAME: ARRIVALS
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
ARRIVALS describes the interarrival distributions for each of the entry queues in the network developed in A233. The interarrivals are assumed to follow the exponential distribution.
DATA TYPE: exponential distribution
SOURCES: A234
DESTINATIONS
INPUT: none
CONTROL: A235
REFERENCE: Appendix B
|
NAME: BOUNDS
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
Bounds is an integer range which represents the upper and lower bounds for the number of processors.
DATA TYPE: INTEGER
MIN VALUE: 1
SOURCES: A21, A213
DESTINATIONS:
INPUT: none
CONTROL: none
REFERENCE: Appendix B
|
NAME: ENTRY TRACE
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The entry trace is a linear list of the entry calls.

DATA TYPE: list
PART OF: PERFORMANCE DATA
SOURCES: A2, A23, A232
DESTINATIONS:
INPUT: none
CONTROL: A236
REFERENCE: Appendix B
|
NAME: INCOMPATIBILITY MATRIX
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The incompatibility matrix is an NxN matrix representing which tasks cannot be scheduled on the same processor.
DATA TYPE: NxN matrix of Boolean
SOURCES: A211
DESTINATIONS:
INPUT: none
CONTROL: A212
REFERENCE: Appendix B
|
NAME: MAXIMAL COMPATIBLE LIST
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
A maximal compatible set is a set of tasks which do not exclude each other from being scheduled on the same processor. This element is a list of all such sets.
DATA TYPE: list
SOURCES: A212
DESTINATIONS:
INPUT: none
CONTROL: A213
REFERENCE: Appendix B
|
NAME: NETWORK
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
Network describes the queueing network developed to represent each of the entries.
DATA TYPE: NxN matrix
MIN VALUE: 0
MAX VALUE: 1
SOURCES: A233

DESTINATIONS:
INPUT: A235
CONTROL: A234
REFERENCE: Appendix B
|
NAME: NON-FUNCTIONAL REQUIREMENTS
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
System constraints expressed in natural language.
DATA TYPE: Natural language
SOURCES: environment
DESTINATIONS:
INPUT: A-0, A1
CONTROL: none
REFERENCE: Appendix B
|
NAME: NUM PROCESSORS
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The number of processors that are available in the system which is being modeled.
DATA TYPE: INTEGER
MIN VALUE: 1
PART OF: RTE
SOURCES: environment
DESTINATIONS:
INPUT: A21, A22, A213
CONTROL: none
REFERENCE: Appendix B
|
NAME: PERFORMANCE DATA
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
Performance criteria is the output of the model and consists of a task schedule, event trace, and performance statistics.
COMPOSITION:
SCHEDULABLE
TASK SCHEDULE
ENTRY TRACE
PERFORMANCE STATISTICS
SOURCES: A-0

DESTINATIONS:
INPUT: none
CONTROL: none
REFERENCE: Appendix B
|
NAME: PERFORMANCE STATISTICS
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The performance statistics are generated from the event trace and the queueing network. The statistics include wait time, service time, queue size, queue utilization, and arrival time.
DATA TYPE: positive, real numbers
PART OF: PERFORMANCE DATA
SOURCES: A2, A23, A236
DESTINATIONS:
INPUT: none
CONTROL: none
REFERENCE: Appendix B
|
NAME: PRECEDENCES
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
Precedences is an NxN matrix which describes if one task is dependent upon another task in order to execute.
DATA TYPE: NxN matrix of Boolean
SOURCES: A231
DESTINATIONS:
INPUT: none
CONTROL: A233, A232
REFERENCE: Appendix B
|
NAME: RTE
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The RTE (runtime environment) consists of the computer hardware and the operating system. The data element contains the number of available processors and the scheduling information.
COMPOSITION:
NUM PROCESSORS
SCHEDULER INFO

SOURCES: environment
DESTINATIONS:
INPUT: none
CONTROL: none
MECHANISM: A-0, A2
REFERENCE: Appendix B
|
NAME: SCHEDULABLE
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
A Boolean flag designating whether the given tasks can be scheduled within the constraints of the task information and RTE.
DATA TYPE: BOOLEAN
VALUES:
FALSE = tasks cannot be scheduled
TRUE = tasks may or may not be schedulable, move onto the next step
PART OF: PERFORMANCE DATA
SOURCES: A2, A21, A213
DESTINATIONS:
INPUT: none
CONTROL: A213
REFERENCE: Appendix B
|
NAME: SCHEDULER INFO
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The scheduler information describes the type of task scheduler and entry scheduler. The entry scheduler is assumed to be a nonpreemptive FCFS scheduler.
PART OF: RTE
SOURCES: none
DESTINATIONS:
INPUT: none
CONTROL: none
MECHANISM: A22
REFERENCE: Appendix B
|
NAME: SOFTWARE DESIGN
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:

The software design is developed prior to invoking the performance model, possibly using a method such as DARTS.

DATA TYPE: Natural language
SOURCES: environment
DESTINATIONS:
INPUT: none
CONTROL: A-0, A1
REFERENCE: Appendix B
|
NAME: SOLUTION
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
Solution represents the solution to the set of simultaneous equations from the queueing network.
DATA TYPE: matrix of equations
SOURCES: A235
DESTINATIONS:
INPUT: none
CONTROL: A236
REFERENCE: Appendix B
|
NAME: TASK INFO
TYPE: DATA ELEMENT
PROJECT: ATM
DESCRIPTION:
The task information is input by the designer and includes the task name or id, period or frequency, execution time, precedence requirements, and entry points.
DATA TYPE: record
COMPOSITION:
TASK_ID
FREQUENCY
EXECUTION_TIME
PRECEDENCES
ENTRY_INFO
SOURCES: A1
DESTINATIONS:
INPUT: A212, A213, A22, A23, A231, A233, A234
CONTROL: A2, A21, A211
REFERENCE: Appendix B
|
NAME: TASK SCHEDULE
TYPE: DATA ELEMENT

PROJECT: ATM
DESCRIPTION:
The task schedule is a schedule based upon the given task information and RTE. The schedule may or may not be optimal.
PART OF: PERFORMANCE DATA
SOURCES: A2
DESTINATIONS:
INPUT: A234
CONTROL: A23
REFERENCE: Appendix B

Appendix C. Validation Programs

C.1 Dining Philosophers Solution

C.1.1 **MACSYMA Batch File.** The r_{ji} probabilities and λs were calculated in MACSYMA. The augmented coefficient λ matrix derived in MACSYMA is shown below.

```
[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1.5 ]
[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0.6 ]
[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0.3 ]
[ 0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0.5 ]
[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0.7 ]
[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0.9 ]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0.6 ]
[ 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0.3 ]
[ 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0.5 ]
[ 0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0.7 ]
[ 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0.9 ]
[ 0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0.1 ]
[ 0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0.2 ]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0.3 ]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0.4 ]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0.5 ]
```

The MACSYMA batch file is shown below.

```
x:matrix([1],[2],[3],[4],[5]);
f0:    1/10;
f1:    x[2] * f0;
f2:    x[3] * f0;
f3:    x[4] * f0;
f4:    x[5] * f0;
l1:    f0 + f1 + f2 + f3 + f4;
x_sum: x[1] + x[2] + x[3] + x[4] + x[5];
r121:  1;
r131:  1;
r141:  1;
r151:  1;
r161:  1;
r12:   x[1]/x_sum;
r13:   x[2]/x_sum;
r14:   x[3]/x_sum;
r15:   x[4]/x_sum;
r16:   x[5]/x_sum;
r23:   x[1]/(x[1] + x[5]);
```

```
r27:   x[5]/(x[1] + x[5]);
r34:   x[2]/(x[1] + x[2]);
r38:   x[1]/(x[1] + x[2]);
r45:   x[3]/(x[2] + x[3]);
r49:   x[2]/(x[2] + x[3]);
r56:   x[4]/(x[3] + x[4]);
r510:  x[3]/(x[3] + x[4]);
r62:   x[5]/(x[4] + x[5]);
r611:  x[4]/(x[4] + x[5]);
r711:  x[5]/(x[1] + x[5]);
r712:  x[1]/(x[1] + x[5]);
r87:   x[1]/(x[1] + x[2]);
r813:  x[2]/(x[1] + x[2]);
r98:   x[2]/(x[2] + x[3]);
r914:  x[3]/(x[2] + x[3]);
r109:  x[3]/(x[3] + x[4]);
r1015: x[4]/(x[3] + x[4]);
r1110: x[4]/(x[4] + x[5]);
r1116: x[5]/(x[4] + x[5]);
R:matrix(
[0,0,0,0,0,0,0,0,0,0,0,r121,r131,r141,r151,r161,11],
[r12,-1,0,0,0, r62,0,0,0,0,0,0,0,0,0,0,0],
[r13,r23,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[r14,0,r34,-1,0,0,0,0,0,0,0,0,0,0,0,0,0],
[r15,0,0,r45,-1,0,0,0,0,0,0,0,0,0,0,0,0],
[r16,0,0,0,r56,-1,0,0,0,0,0,0,0,0,0,0,0],
[0,r27,0,0,0,0,-1,r87,0,0,0,0,0,0,0,0,0],
[0,0,r38,0,0,0,0,-1,r98,0,0,0,0,0,0,0,0],
[0,0,0,r49,0,0,0,0,-1,r109,0,0,0,0,0,0,0],
[0,0,0,0,r510,0,0,0,0,-1,r1110,0,0,0,0,0,0],
[0,0,0,0,0,r611,r711,0,0,0,-1,0,0,0,0,0,0],
[0,0,0,0,0,0,r712,0,0,0,0,-1,0,0,0,0,0],
[0,0,0,0,0,0,0,r813,0,0,0,0,-1,0,0,0,0],
[0,0,0,0,0,0,0,0,r914,0,0,0,0,-1,0,0,0],
[0,0,0,0,0,0,0,0,0,r1015,0,0,0,0,-1,0,0],
[0,0,0,0,0,0,0,0,0,0,r1116,0,0,0,0,-1,0]);
m:echelon(R);
m:subst(m[16]*4/5+m[15],m[15],m);
m:subst(-m[16]*61592/25975+m[14],m[14],m);
m:subst(m[16]*8014/675+m[13],m[13],m);
m:subst(m[16]*491/2355+m[12],m[12],m);
m:subst(m[16]*9/5+m[11],m[11],m);
m:subst(m[15]*77177/20780+m[14],m[14],m);
m:subst(-m[15]*42751/2160+m[13],m[13],m);
m:subst(-m[15]*14/471+m[12],m[12],m);
m:subst(m[15]*7/4+m[10],m[10],m);
m:subst(m[14]*785/108+m[13],m[13],m);
m:subst(m[14]*5/3+m[9],m[9],m);
m:subst(m[13]*6/157+m[12],m[12],m);
m:subst(m[13]*3/2+m[8],m[8],m);
m:subst(m[12]*6+m[7],m[7],m);
```

```
m:subst(m[11]*9/4+m[6],m[6],m);
m:subst(-m[11]*28/27+m[5],m[5],m);
m:subst(-m[10]*15/14+m[4],m[4],m);
m:subst(m[10]*7/3+m[5],m[5],m);
m:subst(-m[9]*6/5+m[3],m[3],m);
m:subst(m[9]*5/2+m[4],m[4],m);
m:subst(-m[8]*2/5+m[2],m[2],m);
m:subst(m[8]*3+m[3],m[3],m);
m:subst(m[7]*6/5+m[2],m[2],m);
m:subst(-m[7]*15/8+m[6],m[6],m);
m:subst(-m[6]*25/3+m[1],m[1],m);
m:subst(m[2]*15+m[1],m[1],m);
m:subst(0,[0],m);
m:1.0*m;
m:subst(1,1.0,m);
m:subst(1.5,[1.5],m);
m:subst(0.1,[0.1],m);
m:subst(0.2,[0.2],m);
m:subst(0.3,[0.3],m);
m:subst(0.4,[0.4],m);
m:subst(0.5,[0.5],m);
m:subst(0.6,[0.6],m);
m:subst(0.7,[0.7],m);
m:subst(0.9,[0.9],m);
quit();
```

C.1.2 **Entry Trace.** A complete entry trace for a 3 meal cycle is shown below:

(Philosopher 0 enters dining room ⟶ Philosopher 1 enters dining room ⟶ Philosopher 2 enters dining room ⟶ Philosopher 0 picks up fork 0 ⟶ Philosopher 3 enters dining room ⟶ Philosopher 1 picks up fork 1 ⟶ Philosopher 2 picks up fork 2 ⟶ Philosopher 3 picks up fork 3 ⟶ Philosopher 3 picks up fork 4 ⟶ Philosopher 3 puts down fork 4 ⟶ Philosopher 3 puts down fork 3 ⟶ Philosopher 2 picks up fork 3 ⟶ Philosopher 3 leaves dining room ⟶ Philosopher 4 enters dining room ⟶ Philosopher 4 picks up fork 4 ⟶ Philosopher 2 puts down fork 3 ⟶ Philosopher 2 puts down fork 2 ⟶ Philosopher 1 picks up fork 2 ⟶ Philosopher 2 leaves dining room ⟶ Philosopher 1 puts down fork 2 ⟶ Philosopher 1 puts down fork 1 ⟶ Philosopher 0 picks up fork 1 ⟶ Philosopher 1 leaves dining room ⟶ Philosopher 3 enters dining room ⟶ Philosopher 3 picks up fork 3 ⟶ Philosopher 0 puts down fork 1 ⟶ Philosopher 0 puts down fork 0 ⟶ Philosopher 4 picks up fork 0 ⟶ Philosopher 0 leaves dining room ⟶ Philosopher 2 enters dining room ⟶ Philosopher 2 picks up fork 2 ⟶ Philosopher 4 puts down fork 0 ⟶ Philosopher 4 puts down fork 4 ⟶ Philosopher 3 picks up fork 4 ⟶ Philosopher 4 leaves dining room ⟶ Philosopher 3 puts down fork 4 ⟶ Philosopher 3 puts down fork 3 ⟶ Philosopher 2 picks up fork 3 ⟶ Philosopher 3 leaves dining room ⟶ Philosopher 2 puts down fork 3 ⟶ Philosopher 2 puts down fork 2 ⟶ Philosopher 2 leaves dining room ⟶ Philosopher 1 enters dining room ⟶ Philosopher 1 picks up fork 1 ⟶ Philosopher 1 picks up fork 2 ⟶ Philosopher 4 enters dining room ⟶ Philosopher 4 picks up fork 4 ⟶ Philosopher 4 picks up fork 0 ⟶ Philosopher 3 enters dining room ⟶ Philosopher 3 picks up fork 3 ⟶ Philosopher 1 puts down fork 2 ⟶ Philosopher 1 puts down fork 1 ⟶ Philosopher 1 leaves dining room ⟶ Philosopher 4 puts down fork 0 ⟶ Philosopher 4 puts down fork 4 ⟶ Philosopher 3 picks up fork 4 ⟶ Philosopher 4 leaves dining room ⟶ Philosopher 3 puts down fork 4 ⟶ Philosopher 3 puts down fork 3 ⟶ Philosopher 3 leaves dining room ⟶ Philosopher 4 enters dining room ⟶ Philosopher 4 picks up fork 4

⟶ Philosopher 4 picks up fork 0 ⟶ Philosopher 0 enters dining room ⟶ Philosopher 2 enters dining room ⟶ Philosopher 2 picks up fork 2 ⟶ Philosopher 2 picks up fork 3 ⟶ Philosopher 4 puts down fork 0 ⟶ Philosopher 0 picks up fork 0 ⟶ Philosopher 4 puts down fork 4 ⟶ Philosopher 0 picks up fork 1 ⟶ Philosopher 4 leaves dining room ⟶ Philosopher 2 puts down fork 3 ⟶ Philosopher 2 puts down fork 2 ⟶ Philosopher 2 leaves dining room ⟶ Philosopher 0 puts down fork 1 ⟶ Philosopher 0 puts down fork 0 ⟶ Philosopher 0 leaves dining room ⟶ Philosopher 1 enters dining room ⟶ Philosopher 1 picks up fork 1 ⟶ Philosopher 1 picks up fork 2 ⟶ Philosopher 1 puts down fork 2 ⟶ Philosopher 1 puts down fork 1 ⟶ Philosopher 1 leaves dining room ⟶ Philosopher 0 enters dining room ⟶ Philosopher 0 picks up fork 0 ⟶ Philosopher 0 picks up fork 1 ⟶ Philosopher 0 puts down fork 1 ⟶ Philosopher 0 puts down fork 0 ⟶ Philosopher 0 leaves dining room)

C.1.3 SLAM II Code.

```
GEN,K. EDWARDS,Dining Philosophers,11/11/90,1,,,,,,,72;
LIM,17,6,5;
NETWORK;
;
; SLAM II IMPLEMENTATION FOR THE DINING PHILOSOPHERS
;
; ATTRIBUTES:
;       ATRIB(1)  = philosopher's seat number
;       ATRIB(2)  = number of meals eaten
;       ATRIB(3)  = eating service time
;       ATRIB(4)  = thinking service time
;       ATRIB(5)  = time philosopher enters dining room
;       ATRIB(6)  = time philosopher begins thinking
;
; BIRTH OF 5 PHILOSOPHERS
;
;       create 5 entities;
;
        CREATE,0.00001,,,5,1;
        ACT/1;
        COLCT,ALL,Birth Times;
        GOON,1;
;
; ASSIGN ATTRIBUTES TO ENTITIES
;
        ASSIGN,ATRIB(1)=NNCNT(1)-1,
               ATRIB(2)=0,
               ATRIB(3)=1.0;
;
; ENTER DINING ROOM
;
ENTER   QUEUE(1);
        ACT,EXPON(0.5);
        ASSIGN,ATRIB(5)=TNOW;
        GOON,1;
        ACT,,ATRIB(1).EQ.0,UPF0; phil 0 will pick up fork 0
        ACT,,ATRIB(1).EQ.1,UPF1; phil 1 will pick up fork 1
```

```
       ACT,,ATRIB(1).EQ.2,UPF2; phil 2 will pick up fork 2
       ACT,,ATRIB(1).EQ.3,UPF3; phil 3 will pick up fork 3
       ACT,,ATRIB(1).EQ.4,UPF4; phil 4 will pick up fork 4
       ACT,,ATRIB(1).GT.5.OR.ATRIB(1).LT.0,ERRHND; error handler
;
; PICKING UP FORKS
;
;  fork 0
;
UPF0   QUEUE(2);
       ACT,DRAND*ATRIB(3);
       GOON,1;
       ACT,,ATRIB(1).EQ.0,UPF1; phil 0
       ACT,,ATRIB(1).EQ.4,DNF0; phil 4
       ACT,,ATRIB(1).NE.0.AND.ATRIB(1).NE.4,ERRHND;
;
;  fork 1
;
UPF1   QUEUE(3);
       ACT,DRAND*ATRIB(3);
       GOON,1;
       ACT,,ATRIB(1).EQ.1,UPF2; phil 1
       ACT,,ATRIB(1).EQ.0,DNF1; phil 0
       ACT,,ATRIB(1).NE.0.AND.ATRIB(1).NE.1,ERRHND;
;
;  fork 2
;
UPF2   QUEUE(4);
       ACT,DRAND*ATRIB(3);
       GOON,1;
       ACT,,ATRIB(1).EQ.2,UPF3; phil 2
       ACT,,ATRIB(1).EQ.1,DNF2; phil 1
       ACT,,ATRIB(1).NE.1.AND.ATRIB(1).NE.2,ERRHND;
;
;  fork 3
;
UPF3   QUEUE(5);
       ACT,DRAND*ATRIB(3);
       GOON,1;
       ACT,,ATRIB(1).EQ.3,UPF4; phil 3
       ACT,,ATRIB(1).EQ.2,DNF3; phil 2
       ACT,,ATRIB(1).NE.2.AND.ATRIB(1).NE.3,ERRHND;
;
;  fork 4
;
UPF4   QUEUE(6);
       ACT,DRAND*ATRIB(3);
       GOON,1;
       ACT,,ATRIB(1).EQ.4,UPF0; phil 4
       ACT,,ATRIB(1).EQ.3,DNF4; phil 3
       ACT,,ATRIB(1).NE.3.AND.ATRIB(1).NE.4,ERRHND;
```

```
;
; PUTTING DOWN FORKS
;
;  fork 0
;
DNF0    QUEUE(7);
        ACT;
        GOON,1;
        ACT,,ATRIB(1).EQ.4,DNF4;  phil 4
        ACT,,ATRIB(1).NE.0.AND.ATRIB(1).NE.4,ERRHND;
        ACT,,ATRIB(1).EQ.0;
        COLCT,INT(5),Phil0 Eat Time;
;       inc meals eaten for phil 0 & assign eating time
        ASSIGN,ATRIB(2)=ATRIB(2)+1,
               ATRIB(4)=9.0;
        ACT,,T0;
;
;  fork 1
;
DNF1    QUEUE(8);
        ACT;
        GOON,1;
        ACT,,ATRIB(1).EQ.0,DNF0;  phil 0
        ACT,,ATRIB(1).NE.0.AND.ATRIB(1).NE.1,ERRHND;
        ACT,,ATRIB(1).EQ.1;
        COLCT,INT(5),Phil1 Eat Time;
;       inc meals eaten for phil 0 & assign eating time
        ASSIGN,ATRIB(2)=ATRIB(2)+1,
               ATRIB(4)=4.0;
        ACT,,T1;
;
;  fork 2
;
DNF2    QUEUE(9);
        ACT;
        GOON,1;
        ACT,,ATRIB(1).EQ.1,DNF1;  phil 1
        ACT,,ATRIB(1).NE.1.AND.ATRIB(1).NE.2,ERRHND;
        ACT,,ATRIB(1).EQ.2;
        COLCT,INT(5),Phil2 Eat Time;
;       inc meals eaten for phil 0 & assign eating time
        ASSIGN,ATRIB(2)=ATRIB(2)+1,
               ATRIB(4)=7/3;
        ACT,,T2;
;
;  fork 3
;
DNF3    QUEUE(10);
        ACT;
        GOON,1;
        ACT,,ATRIB(1).EQ.2,DNF2;  phil 2
```

```
       ACT,,ATRIB(1).NE.2.AND.ATRIB(1).NE.3,ERRHND;
       ACT,,ATRIB(1).EQ.3;
       COLCT,INT(5),Phil3 Eat Time;
;      inc meals eaten for phil 0 & assign eating time
       ASSIGN,ATRIB(2)=ATRIB(2)+1,
              ATRIB(4)=3/2;
       ACT,,,T3;
;
;  fork 4
;
DNF4   QUEUE(11);
       ACT;
       GOON,1;
       ACT,,ATRIB(1).EQ.3,DNF3;  phil 3
       ACT,,ATRIB(1).NE.3.AND.ATRIB(1).NE.4,ERRHND;
       ACT,,ATRIB(1).EQ.4;
       COLCT,INT(5),Phil4 Eat Time;
;      inc meals eaten for phil 0 & assign eating time
       ASSIGN,ATRIB(2)=ATRIB(2)+1,
              ATRIB(4)=1;
       ACT,,,T4;
;
; LEAVE DINING ROOM TO THINK
;
; philosopher 0 thinks
;
T0     ASSIGN,ATRIB(6)=TNOW;
THK0   QUEUE(12);
       ACT,EXPON(ATRIB(4));
       COLCT,INT(6),Phil0 Think Time;
       COLCT,INT(5),Phil0 Cycle Time;
       ACT,,,CYCLE;
;
; philosopher 1 thinks
;
T1     ASSIGN,ATRIB(6)=TNOW;
THK1   QUEUE(13);
       ACT,EXPON(ATRIB(4));
       COLCT,INT(6),Phil1 Think Time;
       COLCT,INT(5),Phil1 Cycle Time;
       ACT,,,CYCLE;
;
; philosopher 2 thinks
;
T2     ASSIGN,ATRIB(6)=TNOW;
THK2   QUEUE(14);
       ACT,EXPON(ATRIB(4));
       COLCT,INT(6),Phil2 Think Time;
       COLCT,INT(5),Phil2 Cycle Time;
       ACT,,,CYCLE;
;
```

```
; philosopher 3 thinks
;
T3      ASSIGN,ATRIB(6)=TNOW;
THK3    QUEUE(15);
        ACT,EXPON(ATRIB(4));
        COLCT,INT(6),Phil3 Think Time;
        COLCT,INT(5),Phil3 Cycle Time;
        ACT,,,CYCLE;
;
; philosopher 4 thinks
;
T4      ASSIGN,ATRIB(6)=TNOW;
THK4    QUEUE(16);
        ACT,EXPON(ATRIB(4));
        COLCT,INT(6),Phil4 Think Time;
        COLCT,INT(5),Phil4 Cycle Time;
        ACT,,,CYCLE;
;
; collect avg cycle time statistics
;
CYCLE   COLCT,INT(5),Avg Cycle Time;
        GOON,1;
        ACT,,ATRIB(2).LT.1000,ENTER;  keep eating
        ACT,,ATRIB(2).GE.1000,DIE;    max meals eaten, exit system
;
; ERROR HANDLER
;
ERRHND QUEUE(17);
        COLCT,ALL,Errors;
        ACT,,,ENTER;
;
; TERMINATE
;
DIE     TERM,5000;
        END;
FIN;
```

C.1.4 **Ada Code.** This section contains the following Ada code for the Dining Philosophers.

- procedure Dining Philosophers
- package Philosopher Info
- procedure Dining
  - task Fork
  - task Host

- task Philosopher
- task Collect Entries
- task Collect Cycle Stats

C.1.5 procedure Dining Philosophers.

```
with Philosopher_Info, Dining, Text_IO;
use Philosopher_Info, Text_IO;
procedure Dining_Philosophers is

   reply        : character;
   Print_Trace  : Boolean := False;

   Num_Meals,
   Maximum_Entries : Integer := 0;

   package Int_IO is new Text_IO.Integer_IO (Integer);

begin

   Text_IO.put_line ("The Dining Philosophers . . .");
   Text_IO.new_line(2);

   -- allows user to input number of meals
   Text_IO.put ("Enter number of meals: ");
   Int_IO.get (Num_Meals);
   Text_IO.new_line;

   -- allow user to turn off entry trace output
   Text_IO.put ("Output the entry trace <y/n>? ");
   Text_IO.get (reply);
   Text_IO.new_line;
   if reply='y' then
      Print_Trace := True;
   end if;

   Maximum_Entries :=  Num_Phils * Entry_Calls * Num_Meals;

   Dining (Maximum_Entries, Num_Meals, Print_Trace);

end Dining_Philosophers;
```

C.1.6 package Philosopher Info.

```
package Philosopher_Info is

   Num_Phils   : constant := 5;
```

```
   Entry_Calls : constant := 6;  -- number of entry calls/cycle

   subtype Phil_Id is integer range 0..Num_Phils-1;

   type Phil_Actions is (enter, leave, up_right_fork, up_left_fork,
                         down_right_fork, down_left_fork);

   subtype Event_String is string (1..35);

   -- These entries correspond to the entry queues
   type Entry_Points is (Enter, Pick_Up_Fork_0, Pick_Up_Fork_1,
                         Pick_Up_Fork_2, Pick_Up_Fork_3,
                         Pick_Up_Fork_4, Put_Down_Fork_0,
                         Put_Down_Fork_1, Put_Down_Fork_2,
                         Put_Down_Fork_3, Put_Down_Fork_4,
                         Think_0, Think_1, Think_2, Think_3,
                         Think_4);

   type Qing_Stat_Record is record
      service       : duration := 0.0;
      wait          : duration := 0.0;
      delta_arrival : duration := 0.0;
      last_arrival  : duration := 0.0;
   end record;

   -- This is a global array
   Qing_Stats : array (Entry_Points) of Qing_Stat_Record;

   function Get_Entry_Index
               (Id     : Philosopher_Info.Phil_Id;
                Action : Philosopher_Info.Phil_Actions)
                                   return Entry_Points;

   function Create_Trace_String
               (Id     : Philosopher_Info.Phil_Id;
                Action : Philosopher_Info.Phil_Actions)
                                   return Event_String;

end Philosopher_Info;

package body Philosopher_Info is

   function Get_Entry_Index
               (Id     : Philosopher_Info.Phil_Id;
                Action : Philosopher_Info.Phil_Actions)
                                   return Entry_Points is
      Index : Entry_Points;

   begin
      case Action is
         when enter => Index := Enter;
```

```
      when up_right_fork =>
         case Id is
            when 0 => Index := Pick_Up_Fork_1;
            when 1 => Index := Pick_Up_Fork_2;
            when 2 => Index := Pick_Up_Fork_3;
            when 3 => Index := Pick_Up_Fork_4;
            when 4 => Index := Pick_Up_Fork_0;
         end case;
      when up_left_fork =>
         case Id is
            when 0 => Index := Pick_Up_Fork_0;
            when 1 => Index := Pick_Up_Fork_1;
            when 2 => Index := Pick_Up_Fork_2;
            when 3 => Index := Pick_Up_Fork_3;
            when 4 => Index := Pick_Up_Fork_4;
         end case;
      when down_right_fork =>
         case Id is
            when 0 => Index := Put_Down_Fork_1;
            when 1 => Index := Put_Down_Fork_2;
            when 2 => Index := Put_Down_Fork_3;
            when 3 => Index := Put_Down_Fork_4;
            when 4 => Index := Put_Down_Fork_0;
         end case;
      when down_left_fork =>
         case Id is
            when 0 => Index := Put_Down_Fork_0;
            when 1 => Index := Put_Down_Fork_1;
            when 2 => Index := Put_Down_Fork_2;
            when 3 => Index := Put_Down_Fork_3;
            when 4 => Index := Put_Down_Fork_4;
         end case;
      when leave =>
         case Id is
            when 0 => Index := Think_0;
            when 1 => Index := Think_1;
            when 2 => Index := Think_2;
            when 3 => Index := Think_3;
            when 4 => Index := Think_4;
         end case;
   end case;
   return Index;
end Get_Entry_Index;

function Create_Trace_String
           (Id     : Philosopher_Info.Phil_Id;
            Action : Philosopher_Info.Phil_Actions)
                                 return Event_String is

   Event : Event_String := (others => ' ');
```

```
   begin
      Event (1..11) := "Philosopher";
      case Action is
         when enter =>
            Event (12..13) := Phil_Id'image(Id);
            Event (14..32) := " enters dining room";
         when leave =>
            Event (12..13) := Phil_Id'image(Id);
            Event (14..32) := " leaves dining room";
         when up_right_fork =>
            Event (12..13) := Phil_Id'image(Id);
            Event (14..27) := " picks up fork";
            if Id/=4 then
               Event (28..29) := Phil_Id'image(Id+1);
            else
               Event (28..29) := Integer'image(0);
            end if;
         when up_left_fork =>
            Event (12..13) := Phil_Id'image(Id);
            Event (14..27) := " picks up fork";
            Event (28..29) := Phil_Id'image(Id);
         when down_right_fork =>
            Event (12..13) := Phil_Id'image(Id);
            Event (14..28) := " puts down fork";
            if Id/=4 then
               Event (29..30) := Phil_Id'image(Id+1);
            else
               Event (29..30) := Integer'image(0);
            end if;
         when down_left_fork =>
            Event (12..13) := Phil_Id'image(Id);
            Event (14..28) := " puts down fork";
            Event (29..30) := Phil_Id'image(Id);
      end case;
      return Event;
   end Create_Trace_String;

end Philosopher_Info;
```

C.1.7 procedure Dining.

```
with Philosopher_Info, Calendar;
procedure Dining (Num_Entries  : in Integer;
                  Num_Meals    : in Integer;
                  Print_Trace  : in Boolean) is

Trace : array (1..Num_Entries) of Philosopher_Info.Event_String;

   task type Philosopher is
      entry Birth (I       : in Philosopher_Info.Phil_Id;
                   t_think : in Float);
```

```
   end Philosopher;

   task type Fork is
      entry Pick_Up (t_accept : out Duration);
      entry Put_Down(t_accept : out Duration);
   end Fork;

   task Host is
      entry Enter (t_accept : out Duration);
      entry Leave (t_accept : out Duration);
   end Host;

   task Collect_Entries is
      entry Next_Entry (Id     : in Philosopher_Info.Phil_Id;
                        Action : in Philosopher_Info.Phil_Actions);
      entry Output_Trace;
   end Collect_Entries;

   task Collect_Cycle_Stats is
      entry Start_of_Day (Id      : Philosopher_Info.Phil_Id;
                          Time    : Duration);
      entry Pass_Timing  (Id      : Philosopher_Info.Phil_Id;
                          t_think,
                          t_eat,
                          t_wait  : duration);
      entry End_of_Day   (Id      : Philosopher_Info.Phil_Id;
                          Time    : Duration);
   end Collect_Cycle_Stats;

   Forks : array (Philosopher_Info.Phil_Id) of Fork;
   Philosophers : array (Philosopher_Info.Phil_Id) of Philosopher;


   task body Philosopher is separate;
   task body Fork is separate;
   task body Host is separate;
   task body Collect_Entries is separate;
   task body Collect_Cycle_Stats is separate;

begin

   Philosophers(0).Birth(0,9.0);
   Philosophers(1).Birth(1,4.0);
   Philosophers(2).Birth(2,2.333);
   Philosophers(3).Birth(3,1.5);
   Philosophers(4).Birth(4,1.0);

end Dining;
```

C.1.8 task Fork.

```
separate(Dining)
```

```
task body Fork is
-------------------------------------------------------------
-- A fork can be picked up or put down.  The Fork task
-- accepts calls to Pick_Up and Put_Down sequentially.
-- t_accept is returned to the calling task and is used
-- to determine the service and wait times for the
-- entry queues.
-------------------------------------------------------------

begin
   loop
      select
         accept Pick_Up (t_accept : out Duration) do
            t_accept := Calendar.Seconds(Calendar.Clock);
         end Pick_Up;
         accept Put_Down(t_accept : out Duration) do
            t_accept := Calendar.Seconds(Calendar.Clock);
         end Put_Down;
      or
         terminate;
      end select;
   end loop;
end Fork;
```

C.1.9 task Host.

```
separate(Dining)
task body Host is

-------------------------------------------------------------
--  The host stands at the door of the dining room and
--  allows the philosophers to enter or leave.
--  Only four philosophers are allowed in the dining
--  room at a time in order to prevent deadlock.
-- t_accept is returned to the calling task and is used
-- to determine the service and wait times for the
-- entry queues.
-------------------------------------------------------------

   Num_in_Room : integer := 0;
   Id          : Philosopher_Info.Phil_Id;

begin
   loop
      select
         when Num_in_Room < 4 =>
            accept Enter (t_accept : out Duration) do
               t_accept := Calendar.Seconds(Calendar.Clock);
            end Enter;
            Num_in_Room := Num_in_Room + 1;
      or
```

```
         when Num_in_Room > 0 =>
            accept Leave (t_accept : out Duration) do
               t_accept := Calendar.Seconds(Calendar.Clock);
            end Leave;
            Num_in_Room := Num_in_Room - 1;
      or
         terminate;
      end select;
   end loop;
end Host;
```

C.1.10 task Philosopher.

```
with Text_IO, Calendar, Random_Number;
use Text_IO, Random_Number;
separate(Dining)
task body Philosopher is

-- Package Random_Number contains the function Next
-- that returns a random float.  The random number
-- generator is used for the eating and thinking delays.

package flt_io is new text_io.float_io(float);

   Eating_Time   : float := 0.9;
   Thinking_Time : float;
   Double        : float := 2.0;

   Left_Fork,
   Right_Fork,
   Temp,
   Id            : Philosopher_Info.Phil_Id;
   Meals_Eaten   : Integer := 0;
   Entry_Index,
   Entry_Index_R : Philosopher_Info.Entry_Points;

   Beg_Think,
   Beg_Eat,
   Beg_of_Cycle,
   End_of_Cycle   : Duration := 0.0;

   q_arrival,
   q_arrival_R,
   q_accept,
   q_accept_R,
   q_complete,
   q_complete_R  : duration := 0.0;

   use Philosopher_Info;
   package Dur_IO is new Fixed_IO (Duration); use Dur_IO;
```

```
begin
   -- get seat assignment and thinking time
   accept Birth (I       : in Philosopher_Info.Phil_Id;
                 t_think : in Float) do
      Id := I;
      Thinking_Time := t_think;
   end Birth;
   Collect_Cycle_Stats.Start_of_Day (Id, Calendar.Seconds(Calendar.Clock));

   -- get fork assignments
   Left_Fork  := Id;
   Temp       := (integer(Left_Fork) + 1) mod 5;
   Right_Fork := Philosopher_Info.Phil_Id(Temp);

   while Meals_Eaten < Num_Meals loop

      Beg_of_Cycle := Calendar.Seconds(Calendar.Clock);

----- enter dining room/collect qing stats/add entry to trace
      q_arrival := Calendar.Seconds(Calendar.Clock);
      Host.Enter(q_accept);
      q_complete := Calendar.Seconds(Calendar.Clock);
      Entry_Index := Get_Entry_Index(Id,enter);
      Qing_Stats(Entry_Index).service :=
         Qing_Stats(Entry_Index).service + (q_complete - q_accept);
      Qing_Stats(Entry_Index).wait :=
         Qing_Stats(Entry_Index).wait + (q_accept - q_arrival);
      if Qing_Stats(Entry_Index).last_arrival /= 0.0 then
         Qing_Stats(Entry_Index).delta_arrival :=
              q_arrival - Qing_Stats(Entry_Index).last_arrival;
      end if;
      Qing_Stats(Entry_Index).last_arrival := q_arrival;

      Collect_Entries.Next_Entry (Id, enter);

----- pick up left fork and collect qing stats and entry trace
      q_arrival := Calendar.Seconds(Calendar.Clock);
      Forks(Left_Fork).Pick_Up(q_accept);
      Entry_Index := Get_Entry_Index(Id,up_left_fork);
      if Qing_Stats(Entry_Index).last_arrival /= 0.0 then
         Qing_Stats(Entry_Index).delta_arrival :=
              q_arrival - Qing_Stats(Entry_Index).last_arrival;
      end if;
      Qing_Stats(Entry_Index).last_arrival := q_arrival;

      Collect_Entries.Next_Entry (Id, up_left_fork);

----- pick up right fork
      q_arrival_R := Calendar.Seconds(Calendar.Clock);
      Forks(Right_Fork).Pick_Up(q_accept_R);
      q_complete_R := Calendar.Seconds(Calendar.Clock);
```

```
      Entry_Index_R := Get_Entry_Index(Id,up_right_fork);
      if Qing_Stats(Entry_Index_R).last_arrival /= 0.0 then
         Qing_Stats(Entry_Index_R).delta_arrival :=
              q_arrival_R - Qing_Stats(Entry_Index_R).last_arrival;
      end if;
      Qing_Stats(Entry_Index_R).last_arrival := q_arrival_R;

      Collect_Entries.Next_Entry (Id, up_right_fork);

   -- eat
      Beg_Eat := Calendar.Seconds(Calendar.Clock);
      delay duration(Eating_Time * Next);
      q_complete := Calendar.Seconds(Calendar.Clock);
      Qing_Stats(Entry_Index).service :=
         Qing_Stats(Entry_Index).service + (q_complete - q_accept);
      Qing_Stats(Entry_Index).wait :=
         Qing_Stats(Entry_Index).wait + (q_accept - q_arrival);

      delay duration(Eating_Time * Next);
      q_complete_R := Calendar.Seconds(Calendar.Clock);
      Qing_Stats(Entry_Index_R).service :=
         Qing_Stats(Entry_Index_R).service + (q_complete_R - q_accept_R);
      Qing_Stats(Entry_Index_R).wait :=
         Qing_Stats(Entry_Index_R).wait + (q_accept_R - q_arrival_R);

----- put down right fork
      q_arrival := Calendar.Seconds(Calendar.Clock);
      Forks(Right_Fork).Put_Down(q_accept);
      q_complete := Calendar.Seconds(Calendar.Clock);
      Entry_Index := Get_Entry_Index(Id,down_right_fork);
      Qing_Stats(Entry_Index).service :=
         Qing_Stats(Entry_Index).service + (q_complete - q_accept);
      Qing_Stats(Entry_Index).wait :=
         Qing_Stats(Entry_Index).wait + (q_accept - q_arrival);
      if Qing_Stats(Entry_Index).last_arrival /= 0.0 then
         Qing_Stats(Entry_Index).delta_arrival :=
              q_arrival - Qing_Stats(Entry_Index).last_arrival;
      end if;
      Qing_Stats(Entry_Index).last_arrival := q_arrival;

      Collect_Entries.Next_Entry (Id, down_right_fork);

----- put down left fork
      q_arrival := Calendar.Seconds(Calendar.Clock);
      Forks(Left_Fork).Put_Down(q_accept);
      q_complete := Calendar.Seconds(Calendar.Clock);
      Entry_Index := Get_Entry_Index(Id,down_left_fork);
      Qing_Stats(Entry_Index).service :=
         Qing_Stats(Entry_Index).service + (q_complete - q_accept);
      Qing_Stats(Entry_Index).wait :=
         Qing_Stats(Entry_Index).wait + (q_accept - q_arrival);
```

```
      if Qing_Stats(Entry_Index).last_arrival /= 0.0 then
         Qing_Stats(Entry_Index).delta_arrival :=
              q_arrival - Qing_Stats(Entry_Index).last_arrival;
      end if;
      Qing_Stats(Entry_Index).last_arrival := q_arrival;

      Collect_Entries.Next_Entry (Id, down_left_fork);

      Meals_Eaten := Meals_Eaten + 1;

----- leave dining room
      q_arrival := Calendar.Seconds(Calendar.Clock);
      Host.Leave(q_accept);
      Entry_Index := Get_Entry_Index(Id,leave);
      if Qing_Stats(Entry_Index).last_arrival /= 0.0 then
         Qing_Stats(Entry_Index).delta_arrival :=
              q_arrival - Qing_Stats(Entry_Index).last_arrival;
      end if;
      Qing_Stats(Entry_Index).last_arrival := q_arrival;

      Collect_Entries.Next_Entry (Id, leave);

      Beg_Think := Calendar.Seconds(Calendar.Clock);
      delay duration(Double * Thinking_Time * Next);
      q_complete := Calendar.Seconds(Calendar.Clock);

      Qing_Stats(Entry_Index).service :=
         Qing_Stats(Entry_Index).service + (q_complete - q_accept);
      Qing_Stats(Entry_Index).wait :=
         Qing_Stats(Entry_Index).wait + (q_accept - q_arrival);

      End_of_Cycle := Calendar.Seconds(Calendar.Clock);

      Collect_Cycle_Stats.Pass_Timing
                          (Id,
                           End_of_Cycle - Beg_Think,
                           Beg_Think - Beg_Eat,
                           Beg_Eat - Beg_of_Cycle);

   end loop;

   Collect_Cycle_Stats.End_of_Day (Id, Calendar.Seconds(Calendar.Clock));

end Philosopher;
```

C.1.11 task Collect Entries.

```
with Text_IO;  use Text_IO;
separate(Dining)
task body Collect_Entries is
```

```
   The_Id     : Philosopher_Info.Phil_Id;
   The_Action : Philosopher_Info.Phil_Actions;

begin
   for i in 1..Num_Entries loop
      accept Next_Entry
        (Id     : in Philosopher_Info.Phil_Id;
         Action : in Philosopher_Info.Phil_Actions) do
         The_Id     := Id;
         The_Action := Action;
      end Next_Entry;
      Trace(i) := Philosopher_Info.Create_Trace_String
                                          (The_Id, The_Action);
   end loop;

      accept Output_Trace;
      if Print_Trace then
         for i in 1..Num_Entries loop
            Text_IO.put_line (Trace(i));
         end loop;
      end if;

end Collect_Entries;
```

C.1.12 task Collect Cycle Stats.

```
with Text_IO;  use Text_IO;
separate(Dining)
task body Collect_Cycle_Stats is

   use Philosopher_Info;
   package Int_IO is new Integer_IO(Integer);
   package Flt_IO is new Float_IO(Float);
   package Clock_IO is new Fixed_IO(Duration);
   package Entry_IO is new Enumeration_IO(Entry_Points);
   use Int_IO, Flt_IO, Clock_IO, Entry_IO;

   Total_Loops : Integer := (Num_Meals + 2)*Num_Phils;
      -- record start and stop times in an array
   type Times is array (Phil_Id) of Duration;
   Start_Times,
   End_Times    : Times;

   -- record timing information for each cycle
   type Timing_Record is record
      think : duration;
      wait  : duration;
      eat   : duration;
   end record;

   type Timing_Array is array (Phil_Id,1..Num_Meals) of Timing_Record;
```

```
   Cycle_Times : Timing_Array;
   Total       : Timing_Record;

   -- keep track of number of times philosophers eat
   type Meals is array (Phil_Id) of Integer;
   Meals_Eaten : Meals := (others => 0);

   Pid    : Phil_Id;
   Cycle_Total,
   Cycle_Subtotal,
   Think,
   Wait,
   Eat    : Duration;

   lambda,
   mu,
   rho,
   W, T, X, N     : float := 0.0;

begin
put_line("total loops = " & integer'image(total_loops));
   -- collect raw data
   for i in 1..Total_Loops loop
put_line ("loop number " & integer'image(i));
      select
         accept Start_of_Day (Id      : Philosopher_Info.Phil_Id;
                              Time    : Duration) do
            Start_Times(Id) := Time;
         end Start_of_Day;
      or
         accept Pass_Timing  (Id      : Philosopher_Info.Phil_Id;
                              t_think,
                              t_eat,
                              t_wait  : duration) do
            Pid   := Id;
            Think := t_think;
            Eat   := t_eat;
            Wait  := t_wait;
         end Pass_Timing;

         Meals_Eaten(Pid) := Meals_Eaten(Pid) + 1;
         Cycle_Times(Pid,Meals_Eaten(Pid)) :=
                     (Think, Wait, Eat);
      or
         accept End_of_Day   (Id      : Philosopher_Info.Phil_Id;
                              Time    : Duration) do
            End_Times(Id) := Time;
         end End_of_Day;

      end select;
```

```
end loop;

-- generate cycle statistics
for i in Phil_Id loop
   Think := 0.0;
   Wait  := 0.0;
   Eat   := 0.0;

   put_line ("Philosopher" & Phil_Id'image(i) & " Average Cycle Times");
   for j in 1..Num_Meals loop

      -- sum the cycle times
      Think := Think + Cycle_Times(i,j).think;
      Wait  := Wait  + Cycle_Times(i,j).wait;
      Eat   := Eat   + Cycle_Times(i,j).eat;
   end loop; -- j index

   Total.think    := Total.think + Think;
   Total.wait     := Total.wait  + Wait;
   Total.eat      := Total.eat   + Eat;
   Cycle_Subtotal := Think + Wait + Eat;
   Cycle_Total    := Cycle_Total + Cycle_Subtotal;

   -- output average cycle statistics
   put ("Thinking Time    = ");
   put (duration(Think/Num_Meals), fore=>6);
   new_line;

   put ("Waiting Time     = ");
   put (duration(Wait /Num_Meals), fore=>6);
   new_line;

   put ("Eating Time      = ");
   put (duration(Eat  /Num_Meals), fore=>6);
   new_line;

   put ("Delta Cycle Time = ");
   put(End_Times(i) - Start_Times(i), fore=>6);
   new_line;
   put ("Actual Time      = ");
   put (Cycle_Subtotal, fore=>6);
   new_line;
   put ("Overhead         = ");
   put (End_Times(i) - Start_Times(i) - Cycle_Subtotal, fore=>6);
   new_line(2);
end loop;  -- index i

put_line ("Averages for all the Cycles");
put ("Thinking Time = ");
put (duration(Total.think/(Num_Phils * Num_Meals)));
new_line;
```

```
      put ("Waiting Time  = ");
      put (duration(Total.wait/(Num_Phils * Num_Meals)));
      new_line;
      put ("Eating Time   = ");
      put (duration(Total.eat/(Num_Phils * Num_Meals)));
      new_line;

      -- output qing stats
      new_line(2);
      put_line ("             Queueing Timing Statistics "); new_line;
      put_line ("                      Delta");
      put_line ("Entry Point        Arrival Time   Service Time   Wait Time");
      put_line ("-----------        ------------   ------------   ----------");
      for i in Entry_Points loop
         put (i, width=>18);
         put (duration(Qing_Stats(i).delta_arrival/(Num_Meals-1)), fore=>5);
         put (duration(Qing_Stats(i).service/Num_Meals), fore=>10);
         put (duration(Qing_Stats(i).wait/Num_Meals), fore=>10);
         new_line;
      end loop;

      N := float(Num_Meals);

      for i in Entry_Points loop
         X := float(Qing_Stats(i).delta_arrival);
         lambda := (N-1.0)/X;
         X := float(Qing_Stats(i).service);
         mu := N/X;
         rho    := lambda / mu;
         W      := (rho/mu)/(1.0-rho);
         T      := (1.0/mu)/(1.0-rho);

         new_line;
         entry_IO.put (i, width=>18); new_line;
         put ("     lambda = "); flt_IO.put(lambda, exp=>0,aft=>4); new_line;
         put ("     mu     = "); flt_IO.put(mu,     exp=>0,aft=>4); new_line;
         put ("     rho    = "); flt_IO.put(rho,    exp=>0,aft=>4); new_line;
         put ("     W      = "); flt_IO.put(W,      exp=>0,aft=>4); new_line;
         put ("     T      = "); flt_IO.put(T,      exp=>0,aft=>4); new_line;
      end loop;
      Collect_Entries.Output_Trace;

   end Collect_Cycle_Stats;
```

Bibliography

1. ACM Special Interest Group on Ada (SIGAda), Ada Runtime Environment Working Group (ARTEWG). *A Framework for Describing Ada Runtime Environments.* ACM SIGAda ARTEWG, 15 October 1987.
2. Bailor, Maj Paul D. Class handout distributed in CSCE 693, Principles of Embedded Software. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 2 January 1990.
3. Booch, Grady. *Software Engineering with Ada* (Second Edition). Menlo Park CA: Benjamin/Cummings Publishing Company, Inc., 1987.
4. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design, A Foundation.* Addison-Wesley Publishing Company, 1988.
5. Coffman, Edward G. and Peter J. Denning. *Operating System Theory.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1973.
6. Cross, Dr Joseph K. *The Ada Run-time Environment.* Sperry Univac, Defense Systems Division, P.O. Box 3525, St. Paul MN 55164-0525, 1984.
7. Department of Defense. *Computer Programming Language Policy.* DOD Directive 3405.1. Washington: Government Printing Office, 2 April 1987.
8. Department of Defense. *Military Standard: Language Reference Manual for the Ada Programming Language - ANSI/MIL-STD-1815A.* Washington: Government Printing Office, January 1983.
9. Emshoff, James R. and Roger L. Sisson. *Design and Use of Computer Simulation Models.* New York: Macmillan Publishing Co., Inc., 1970.
10. Gehani, Narain. *Ada Concurrent Programming.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1984.
11. Gomaa, Hassan. "A Software Design Method for Distributed Real-Time Applications," *The Journal of Systems and Software, 9:* 81-94. (1989).
12. Hartrum, Thomas C. *$IDEF_0$ Requirements Analysis, EENG 593.* School of Engineering, Air Force Institute of Technology (AU), Wright- Patterson AFB OH, 2 January 1990.
13. Hoare, C.A.R. *Communicating Sequential Processes.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1985.
14. Hoare, C.A.R. "The Emperor's Old Clothes," *Hard Real-Time Systems,* edited by Stankovic, John A. and Krithi Ramamritham. Computer Press of the IEEE, 1730 Massachusetts Avenue, N.W., Washington, D.C., 20036-1903.
15. Ichbiah, Jean D. and others. *Rationale for the Design of the Ada Programming Language.* Ada Joint Programming Office, United States Department of Defense, Washington DC 20301-3081, 1986. (AD-A187 106).

16. Kleinrock, Leonard. *Queueing Systems, Volume 1: Theory.* New York: John Wiley & Sons, Inc., 1975.

17. Kleinrock, Leonard. *Queueing Systems, Volume 2: Computer Applications.* New York: John Wiley & Sons, Inc., 1976.

18. Milenkovic, Milan. *Operating Systems, Concepts and Design.* New York: McGraw-Hill Book Company, 1987.

19. Peterson, James L. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1981.

20. Pritsker, A. Alan B. *Introduction to Simulation and SLAM II* (Third Edition). New York: John Wiley & Sons, Inc., 1986.

21. Seward, Capt Walter D. *Optimal Multiprocessor Scheduling of Periodic Tasks in a Real-Time Environment.* PhD dissertation. School of Engineering, Air Force Institute

of Technology (AU), Wright-Patterson AFB OH, December 1979.

22. Sha, Lui and John B. Goodenough. *Real-Time Scheduling Theory and Ada* Technical Report CMU/SEI-89-TR-14. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

23. Silberschatz, Abraham and James L. Peterson. *Operating System Concepts* (Alternate Edition). Addison-Wesley Publishing Company, Inc., 1988.

24. Stankovic, John A. and Krithi Ramamritham. *Hard Real-Time Systems.* Computer Press of the IEEE, 1730 Massachusetts Avenue, N.W., Washington, D.C., 20036-1903.

25. Trivedi, Kishor Shridharbhai. *Probability & Statistics with Reliability, Queuing, and Computer Science Applications.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1982.

26. Whitted, Capt Gary Alen. *Determination of the Underlying Task Scheduling Algorithm for an Ada Runtime System.* MS Thesis, AFIT/GCS/ENG/89D 18. School of Engineering of the Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE
FEASIBILITY ANALYSIS OF DEVELOPING A FORMAL PERFORMANCE MODEL OF Ada TASKING

5. FUNDING NUMBERS

6. AUTHOR(S)
Kathryn J. Edwards, GS-12, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/90D-18

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Language Evaluation Center
ASD/SCOL
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As software system requirements become more complex, software engineers must carefully design the systems to ensure the systems adequately meet all the requirements, both functional and non-functional. Because real-time systems have timing constraints, in addition to the more traditional behavioral constraints, a comprehensive software design analysis model is required which incorporates performance, timing, and behavioral constraints. Although the Ada language tasking constructs are compiler independent, Ada tasking is dependent on its run-time environment; therefore, a formal model of Ada tasking and its associated runtime environment is important in order for system designers to make realistic decisions when modeling Mission Critical Computer Resources (MCCR) systems. The main focus of this research effort is to determine the feasibility of developing a parameterized, formal model of Ada tasking and the associated runtime environment. This research shows that such a parameterized model can be developed using a mathematical model which incorporates real-time scheduling and queueing theory. This model can be used in the future to develop a design analysis environment for real-time embedded software systems that require Ada as the target language. Thus, given a specification for such a system, the design analysis environment can be used to obtain the information needed to support Ada software design decisions.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada Tasking, Real-Time Systems, Formal Model, Mathematical Models, Queueing Theory, Ada Runtime Environment, Software Design Analysis	124
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered
State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as. Prepared in cooperation with...; Trans. of..., To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- DOE - See authorities.
- NASA - See Handbook NHB 2200.2.
- NTIS - Leave blank.

Block 12b. Distribution Code.

- DOD - Leave blank.
- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- NASA - Leave blank.
- NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

*U.S.GPO:1990-0-273-271